Plant Pathology 2020 – FGVC7

Authors:

Julen Arizaga

Daniel Suarez

Content

Figures

Tables

Problem statement

In this project we will implement the approach of the ELoPE **[1]** paper in order to obtain the best possible position in the Plant Pathology 2020 - FGVC7 **[2]** Kaggle competition. In this Kaggle competition we are given images of plants, which contains 1,821 1800 x 2100 color images from 4 different classes: healthy, scab, rust or multiple diseases.

The test set contains another 1,821 images, and the metric used to evaluate this test set is going to be the Mean Columnwise Area Under Receiver Operating Caracteristic Curve.

Proposed solution

In order to build the best possible model, several steps have to be taken, from data visualization and handling to hyperparameter tuning. Here the path taken to arrive to the final model will be discussed, as well as the problems encountered during this path.

Visualization

The first step is to visualize and examine the data, in order to have an understanding of it and understand better the problem we have to handle.

Class identification

In order to see which properties each class has, we will take one image of each class and print them, in order to draw conclusions. Here there are the images.

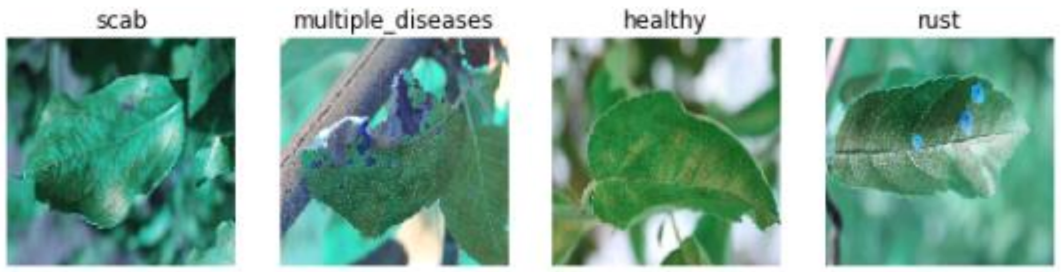

Figure 1: Plant images with categories

As it can be observed the rust, scab and multiple_diseases classes have some kind of stain or are eaten by something. In order to know more about the problem at hand, further research has been done about those diseases, which are caused by fungi:

The most obvious symptoms of Scab occur on leaves and fruit in the spring and summer, and look like very small velvety brown to olive-green spots that enlarge and darken to become more or less circular.

Rust can be identified early on by white, slightly raised spots on the undersides of leaves and on the stems. After a short period of time, there spots become covered with reddish-orange spore masses. Later, leaf pustules may turn yellow-green and eventually black. Severe infestations will deform and yellow leaves and cause leaf drop. As it can be observed rust has many stages, and each of them is different both in color and in magnitude, so it can be difficult to identify it.

The multiple diseases class in this dataset corresponds to plants that have rust and scab and possibly more diseases. Therefore those plants will have the properties of both rust and scab classes.

Class balance

In this case, as it has been previously mentioned, there are 4 classes, but those 4 classes don't have the same representation. The data is unbalanced, and its distribution can be seen in the figure 2

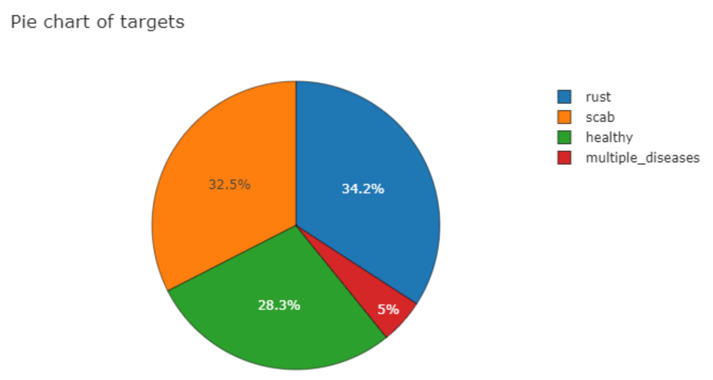

Figure 2: CLass balance chart

From the 4 classes, there are 3 with high and more or less equal representation, but the "multiple_diseases" class has a much lower representation. Therefore, in order to have an equal representation in the training and validation set, we will split the data with a stratify function.

Data Handling

Now we will define the process done with the images in order to be ready to fit them into the model. This part constitutes of how the train/validation split has been done, together with the data augmentation performed.

Train/Validation/Test split

Here we will discuss how we have done the data split.

As we have previously mentioned we have 1821 labeled images, which aren't much taking into account that we have 4 classes, and the amount of images from the "multiple_diseases" class is 91 (really few images). Nevertheless, we know this class has the attributes of "rust" and "scab", as it usually has those two diseases, therefore if the final prediction probability of scab and rust is high, we could conclude that the image has multiple diseases.

In order to do the data split, we have to take into account that we don't have much data. We want to give most of this data to the training set, but we are also going to tune a lot of hyperparameters and try different architectures with this data, so we are going to need a robust validation set in order to decide which model is the best. Therefore, we will set a 80/20 partition for the train/validation set, and as we will have only about 1,457 images in the training set, we will perform synthetic data augmentation in order to train the model with more images and not to overfit the model too.

About the test set, we don't have to do anything here as this set is given by the competition. This set contains another 1,821 images of the same size as the training set, which means it is big enough to certainly check if our model is doing correctly or not. The only drawback here is that we don't have the labels, so we cannot measure the performance with the metric we want. The performance will be measured with Mean Columnwise Area Under Receiver Operating Caracteristic Curve as said before, and we will finally choose the model that is closer to 1 on this metric.

Data Augmentation

As it has been previously said, in order to augment the number of images we have in the training set (as we don't have many images) we will create synthetic images. We will apply the following techniques to the original images:

- Rotation: We will rotate the images up to 45 degrees
- Shift: We will shift the images up to a 20%
- Flip: We will flip the images both horizontally and vertically

Here we have an example of flipping and rotation.

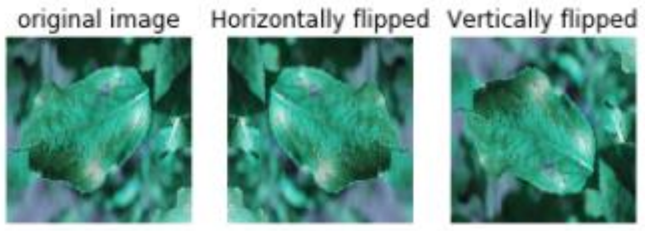

Figure 3: Rotated imaged

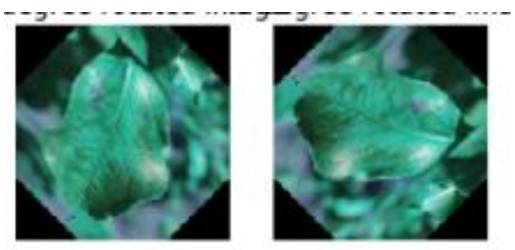

Figure 4: Flipped images

As the augmented and normalized images (normalized they occupy 4x more) can take up to 8GB in memory, and this, together with the model, makes it too much for both our computer and the Xsede cluster, we have decided to feed it by a Data Generator. The benefits of doing this is that we don't have to store all the images in memory, only the ones that are going to be used in each batch, and as the data augmentation is performed randomly (between specified limits), we will have slightly different images in each epoch, which will prevent our model from overfitting. The main drawback from this method is the loss of time, it will take much longer since the images have to be fetched from disk each time they want to be used.

Paper Selection

We have selected two papers in order to solve this problem as effectively as possible. The first paper **[3]** relies on a new Loss implementation called Batch Confussion Norm, whose main idea is that all the images from the batch are from different classes. This is usually useful and true for problems with a lot of classes, such as bird species classification where we could have up to 1000 classes and the batch size is usually much lower, therefore it is reasonable to think that we won't have two images of the same class in the same batch.

Nevertheless, this approach was not useful for us, as we only have 4 classes, it is not reasonable to think that in each batch we have different classes, even if our batch is of 4 or 2. Therefore, after implementing the Batch Confussion Norm approach and training a model, we have decided to use the other papers implementation.

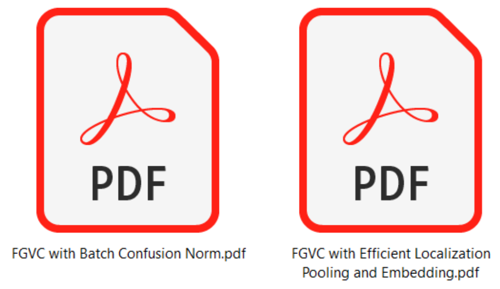
PDF
PDF
FGVC with Batch Confusion Norm.pdf
FGVC with Efficient Localization Pooling and Embedding.pdf

Figure 5: Different Papers

We have also taken into account the second part of this paper, which is an atrous convolution at the end of the CNN backbone, but it introduced much more parameters into our model, which made it harder to train. Also, the implementation of this atrous convolution was not compatible with the second papers approach, therefore we have not used it.

Paper implementation

The paper used to make the implementation has been EloPe **[1]**. State-of-the-art approaches typically involve a backbone CNN such as ResNet or VGG that is extended by a method that localizes and attends to specific discriminative regions. This paper aims to improve the performance of a given backbone CNN with little increase in complexity and requiring just a single pass through the backbone network. Specifically, the following three steps are proposed:

- **Localization module**: In order to avoid having to rely on human bounding box annotations, an efficient bounding box detector is trained and applied before the image is processed by the backbone CNN. This localization module is lightweight and trained using only the class labels.

- **Global k-max pooling**: This two-step pooling procedure first applies k-max pooling at the last convolutional layer which is followed by an averaging operation over the K selected maximal values in each feature map. This could be seen as a very simple form of attention.

- **Embedding layer**: An embedding layer is inserted into the backbone CNN as penultimate layer, which is trained with a loss function composed of two parts: within-class loss and between-class loss

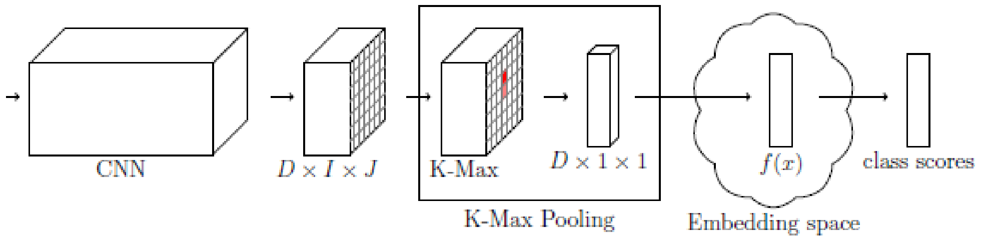

Figure 6: Paper implementation

Now we will explain how those steps have been implemented.

Localization module

As it is stated in the paper, the localization module is equivalent to the first few layers of a ResNet-50 (initialized from the trained classification model). Therefore, as the increase of performance due to the independent localization module is not very high and the Embedding layer implementation has taken longer than expected, we have relied on the main CNN backbone to perform this localization.

Global k-max pooling

In order to implement the new K-max Pooling layer in Keras, we have made a new class named "GlobalKMaxPooling2D", which inherits the properties of Keras' Layer class. This new class' structure is essentially the same as GlobalMaxPooling2D from Keras, but it differs in the call function and input arguments.

As input arguments we have to add the "K", where K is the number of activations we want to take into account from the last channel's output. It is specified to be 10 if nothing is said, but we have done our model to input K = 4, as it is stated in the paper that they got the best results with this number.

In the call function, where the main functionality is, the input has been reshaped to have all the activations in one dimension (instead of 2 as previously was), we have applied a function to get the top k activations from this new dimension and we have calculate their mean.

In order to make it more visual, this is the function that has been developed:

$$GKMP(y)_d = \frac{1}{K}\sum_{k=1}^{K} S_{d,k} \tag{1}$$

Where d is the number of output channels, S are the activations sorted in descending order and K is the number of activations we want to make the mean of.

Embedding layer

In order to calculate the within class loss and between class loss we need the output of the Global k-max pooling layer, and as Keras custom functions only allow us to take the real labels and the predicted ones as input, we have been forced to create a new class, in which we have developed the model, as well as done other functions to save it. This way, we have been able to pass the output of this layer to the loss functions by specifying the layer's output as a class property.

As the implementation had to be developed in tensorflow, and in order to be efficient no for loop or numpy implementation could be used, we have implemented both functions by playing with the dimensions. We have expanded the dimensions of the vectors, applied some masks and done operations between them. As it is hard to explain how it has been developed in tensorflow, we will keep it simple and just explain what do those losses do, and how they make it easier for the algorithm to generalize properly.

Within-class loss

For both losses we will keep track and calculate each function's mean (after passing the global k-max pooling). In the within-class loss, we will penalize the images whose output is very different from the calculated mean from the previous images, therefore enforcing all the outputs of the same class to have similar values. In summary, what we want is to enhance the within-class similarity, and penalize the weights that make same class images look like different.

The equation to do it is the following:

$$L_w = \frac{1}{2N} \sum_{n=1}^{N} \| f(x_n) - \mu_{c_n}^{t} \|_2^2 \tag{2}$$

The "u" from this equation is the mean of the class to which x belongs, and f(x) is the output of the global k-mas pooling of the image. Finally, N is the number of images in the batch.

Between-class loss

In the between-class loss we want to make the outputs from the different classes as different as possible. In order to do so, we will penalize the class similarity above a threshold. The equation we are going to use is the following:

$$L_b = \frac{\gamma}{4|P|} \sum_{(k,c)\in P} \max\left(m - \|\mu_k^t - \mu_c^t\|_2^2, 0\right)^2 \quad (3)$$

Here, the set P will have all the possible combinations of different classes, and the difference between all this classes will be computed. If the norm of the difference between those classes is lower than a specified threshold "m", we will have loss, otherwise, this loss will be 0. With this max function we make the loss null when the different classes' output are different, and give it a value when they are more similar. Therefore we don't have learning due to this loss until the classes mean are more similar than the specified threshold.

Path to best model

Once we have developed all the code to implement the paper, we have to tune the hyperparameters in order to obtain the best possible model. This is no easy task as, apart from the typical hyperparameters of a neural network (which are a lot), now we also have the hyperparameters of the loss functions.

At the beginning, we thought that the loss function hyperparameters could be the same as the ones in the paper, and even if for most of them this is true, some parameters have had to be highly tuned.

We also have faced several problems and errors with our implementation, which seemed to be okey at the beginning, but we have seen there were some errors. Those errors will be commented later, and now we will focus on how we have developed our first model.

First approach: set baseline

In order to set a baseline and see how well our model can fit the data, the first thing we have done is train the model with the data we have for 20 epochs, before it gets overfitted. Here is where we obtain the maximum validation accuracy, and the metrics are the followings:

Training categorical accuracy	Validation categorical accuracy
0.9121	0.8967

Table 1: Baseline accuracy

As it can be seen, the training accuracy is higher than the validation accuracy (which is normal), but the gap isn't too large to say it is overfitted.

First question: Can our model fit the data?

The first question we ask ourselves was if the model could fit our data. Before tuning the hyperparameters in order to get the best model with best validation accuracy, we have to make sure that our model has enough parameters and can overfit our data. In order to do this, we have run our model for 40 epochs, without data augmentation, and those are the results:

Training categorical accuracy	Validation categorical accuracy
0.9923	0.8165

Table 2: Overfitting accuracy

As it can be inferred from this result, our Backbone can overfit the data, and therefore it is complex enough.

Path to go: hyperparameter tuning

In order to build the best mode, the only thing we have to do right now is choose the right parameters in order to do this, which seems easy, but really isn't. Usually, the best way to choose the hyperparameters would be to use some kind of software such as hyperas in order to get the optimum hyperparameters, but in this case we haven't relied on software because of the following reasons:

- Too many hyperparameters: There are too many hyperparameters in order to regularize the model, and each of them will have a different impact on the final model. The hyperas software doesn't know the meaning of all the hyperparameters, but we do, so we can play with them more accurately.
- Errors during execution: As we were training models, we realized there were errors in the code, so we had to begin again "from scratch" in which refers to the hyperparameter tuning.
- Time constraints: The main issue has been the time needed in order to run one model. Each model took about 4 hours to complete, so in order to make 6 iterations we would need a whole day. Xsede only allows us to reserve 48 hours, which means we could make 12 iterations.

Therefore, because of errors handled during execution and time constraints, we have relied on ourselves in order to do the hyperparameter tuning.

Hyperparameter tuning

As said before, we have used our understanding of the model and the hyperparameters in order to tune the model. Those are the hyperparameters we have tuned:

- Image size
- Data Augmentation
- Batch size
- Landa
- Learning rate
- CNN backbone
- Layer freezing

Image size

We have tried different image sizes, from 224 to 448 because of the memory issues and the time needed to compute it. We have seen that the image seemed not to have a high impact on the final accuracy of the model, but when doing the rest of the hyperparameter tuning we have seen setting image size of 448 x 448 improved the final accuracy 1% over images of size 224 x 224.

Data Augmentation

About data augmentation, we have done it with the ImageGenerator, but we have used the parameters to do the data augmentation as a regularization parameter. We have observed that the lower those parameters were, the easier the model got overfitted, and if they were too high (too much zoom or rotation for example), the model didn't fit well all the data. Therefore, we have tried different approaches till reaching the one that did best on our model.

Batch size

The batch size has ben an easier to tune hyperparameter. Because of the time constraints we needed a low batch size, because otherwise each model would take about 8 hours or more in order to be completely trained. Therefore, after ranging from 2 to 16, we have decided to choose a batch size of 4, as it was the best one taking into account both accuracy and time.

Landa

The landa hyperparameter has been also hard to train, as in the paper was stated to be 2, but for our application we have had to set it very low, under 0.01. This change in this hyperparameter is because of the class means, as they are squared to obtain the within class loss, it seems our model has much bigger class means than the one from the paper. Therefore, we have used a wide range of landa parameters, using this also as a regularization hyperparameter. The higher the landa, the more value we give to within class loss and between class loss, and therefore the model generalizes better.

Learning rate

The learning rate has been difficult to choose. The adam optimizator seemed not to choose the learning rate well, as we obtained high variance when the model was nearly fitted. Therefore we have had to came up with a learning rate schedule, which in our case increases linearly until the 5th epoch and then decreases exponentially.

CNN backbone

The backbone chosen in the paper was the Resnet50, but as nowadays there are a lot of pre-trained backbones, we have tried a lot of different other backbones, such as EfficientNetB5, Xception and InceptionResNetV2. There has been memory issues, and therefore we were not able to try EfficentNetB7, but finally the most accurate backbone has shown to be the Xception.

Layer freezing

We have tried different ways of layer freezing and unfreezing, but this seemed not to work for us. Freezing the entire backbone was not a option for us, as the within class loss was too high (the output of the global k-max pooling couldn't be trained), so we tried freezing part of the backbone, training, unfreezing some of it and so, but this seemed not to work totally well either. Finally, it has been shown that the best approach was to train the entire network from the beginning, so we have followed this method.

Implementation details

Here we will discuss the problems and issues we have had with the implementation.

Loss functions

Then main issue we have had about programming has been to develop the two loss functions specified in the paper. This has been a tough job, as they have to be implemented in TensorFlow and without numpy functions, in order to be able to be executed in graph mode (for the best performance when training). In order to implement those functions in tensorflow without the need of any for loop, we have developed an algorithm that expands the vectors, multiplies them, applies masks and shortens them in order to perform a loop in a vectorized way.

Figure 7: Tensorflow logo

This was no easy task and therefore we had to develop a method in order to know if our implementation was right or not. In order to do this we developed the same algorithm in numpy (we checked this one by doing a proof by hand), and finally we got the same results for the same input. This, once more, was an iterative process in which we saw many places where our first implementation was wrong, as we didn’t have experience with Tensorflow.

Image generator vs. previous loading

In order to feed the images to train the model, there are two approaches: Use an image generator and put the images into disk as they are needed or load all the needed images into memory before feeding them. Both of them have their advantages and disadvantages, and here we will present them

Previous loading

This is the easiest and fastest way of doing it, but it requires a lot of memory, even more if we do data augmentation. This is the first approach that we took, as we wanted the fastest approach, but when we normalized the images we realized that they didn’t fit into memory. The images are loaded in a format wehere each pixel occupies 8 bytes, but as we convert them into float32, they occupy 32 bytes, which is 4x more. Although this might not seem a problem, in our case it raised the needed memory from 2GB to 8GB of ram, which we couldn’t afford if we wanted to train the model also.

Image generator

This is finally the approach that has been taken, as we couldn’t afford to have all the images in memory. In memory terms this is the best method, but it takes quite longer than the previous one, as the images are in disk and they have to be fetched.

Apart from memory, another positive aspect of this method is that the image augmentation is done randomly when the images are loaded, and therefore, we get slightly different images in each epoch, which helps our model generalize better.

Instructions to run the code

Our code contains five files. In those five files we implement both papers, but do the data handling necessary only for the EloPE paper.

Batch_Confusion_Norm.ipynb

This Python Notebook implements the Batch Confusion Norm paper. In order to do this the images are loaded, normalized and fed to the model in order to train. Finally, when the model is trained the weights are saved.

Data_handling.ipynb

This Python Notebook does the data augmentation and stores it in disk. This was used in a first approach but not later, so if the user wants to use the data generator as we have done, he/she shouldn't execute this file. Otherwise, if the user wants to store all the augmented images in disk, to load them later and fed to the model, execute this file paying special attention to the directories.

Generate_folders.ipynb

This Python Notebook takes the images from the training set and stores them in different folders. This Norebook also does the data partition. It has to be executed if the user wants to used data generators flowing from directory as we have done.

Elope_V5.py

This is the script that implements the main functionality of the project. It is a Python script (in order to run it in an external server) that has the implementation of the paper, together with methods to load the data in different ways, save the best model (with a callback) and print and save the plots of the loss and different metrics we have used. This is the script that has been modified in order to find the best model, and the parameters of the best model are inside it.

Predictions.ipynb

This Python Notebook implements the necessary functions to read the previously trained models and the test dataframe, together with the testing images and predict their label. Finally those labels are put in the test dataframe and it is saved. Finally, we have made an ensembling, in order to combine our best model together with previous models in order to get a better score in Kaggle.

Results and discussion

After walking all the path and doing the innumerable iterations in order to set the right hyperparameters, here we will present our results and we will discuss the improvements the implementation of the paper brings.

Results

After doing a lot of iterations in order to get the best hyperparameters, we have chosen the following ones:

- **Image size** = 448 x 448
- **Data Augmentation** -> rotation_range = 30, width_shift_range = 0.2, height_shift_range = 0.2, shear_range = 0.2, zoom_range = 0.2, horizontal_flip = True, vertical_flip = True, fill_mode = 'nearest'
- **Batch size** = 4
- **Landa** = 0.0003
- **Learning rate** -> from 0.00001 to 0.00005 linearly (till 5th epoch), and thin decrease till 0.00001 exponentially (exponent = 0.8)
- **CNN backbone** = Xception
- **Layer freezing** = None

With this method we have obtained the following results for the best epoch:

Training categorical accuracy	Validation categorical accuracy
0. 9746	0. 9616

Table 3: Best result accuracy

Those are the graphics obtained:

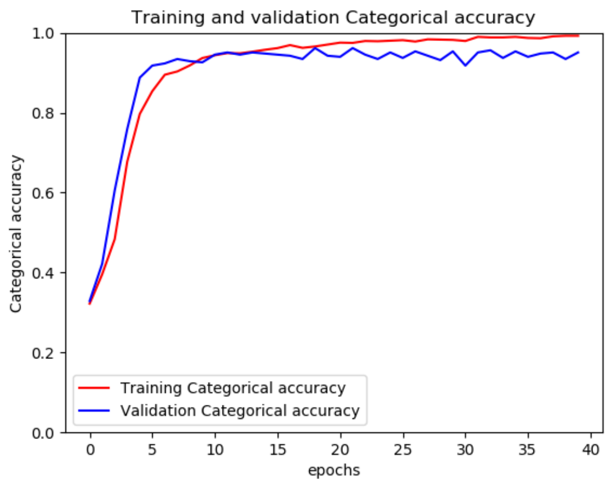

Figure 8: Best model accuracy

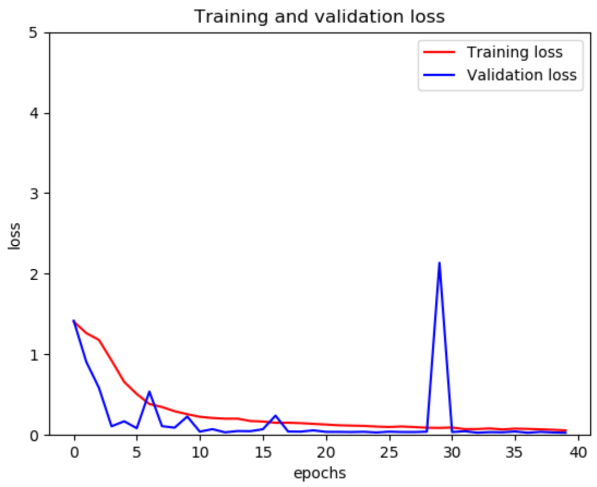

Figure 9: Best model loss

As it can be seen, our best model obtains a 96%+ accuracy on the validation set, which is a high improvement of the baseline model. The model behaves slightly better in the training set than in the validation set, and it gets slightly overfitted after the epoch number 30, though this cannot be appreciated in the figures.

The training loss decreases continuously, getting to near 0 in the end, and the validation loss decreases much faster than the training one, though it has much more variance and has a strange peak in the 30th epoch.

Results without paper implementation

In order to discuss the improvement in performance the implementation of the paper gives to our model, we have made an hyperparameter tuning in order to obtain the best possible model without the paper implementation, and we have obtained the following for the best epoch:

Training categorical accuracy	Validation categorical accuracy
0.9535	0.9370

Table 4: Without paper best model accuracy

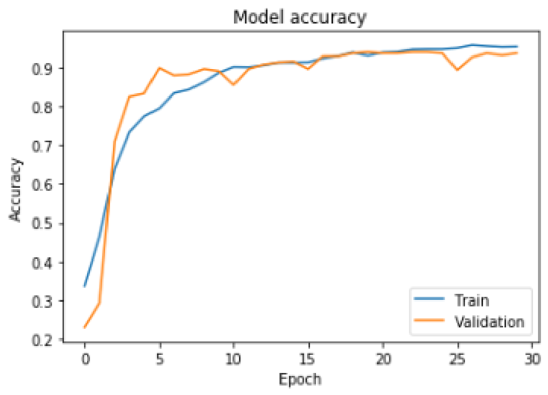

Figure 10: Accuracy without implementation

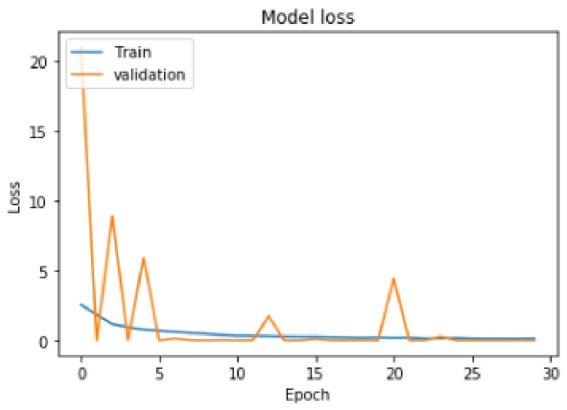

Figure 11: Loss without implementation

As it can be appreciated both in the accuracy and the loss plot, the validation set seems to have more variance than the training set, which is totally normal, but they both have consistent results and reach a good accuracy without being overfitted.

Discussion

Here we will discuss if the paper implementation improved the model or not. Here we have the comparison between the best model obtained with and without the paper:

	Without paper	With paper
Training cat. accuracy	0.9535	0.9746
Validation cat. accuracy	0.9370	0.9616
Error in val.	6.3%	3.8%
Kaggle score	0.924	0.957

Table 5: Summary of performance

As it can be observed, with the paper implementation we have lowered the error in the validation set to nearly the half of the one obtained without the paper. Therefore we can conclude the implementation of the papers method has improved greatly the accuracy of our model.

About the Kaggle competition, we got a good score, but got stuck in the top 52%. We have seen other notebooks and in none of them a validation score of more than 0.95% was obtained. Therefore the gap for improvement of our model is surely in other facts, such as image preprocessing or data handling.

References

[1] ELoPE: Fine-Grained Visual Classification with Efficient Localization, Pooling and Embedding, Harald Hanselmann and Hermann Ney, 17 Nov 2019

[2] Plant Pathology 2020 – FGVC7. Retrieved from https://www.kaggle.com/c/plant-pathology-2020-fgvc7/overview

[3] Fine-Grained Visual Classification with Batch Confusion Norm, Yen-Chi Hsu, Cheng-Yao Hong, Ding-Jie Chen, Ming-Sui Lee, Davi Geiger, Tyng-Luh Liu, 28 Oct 2019

[4] Tensorflow documentation. Retrieved fromhttps://www.tensorflow.org/api_docs/python/

[5] StackOverflow answers. Retrieved from https://stackoverflow.com/