
Artificial Neural Networks and Deep Learning

# Classification of plants using CNN

*Authors:*
La Greca Michele Carlo (10864460)
Giuffrida Andrea (10643540)
Nicolis Nicholas (10867841)

*Professors:*
Prof. M. Matteucci
Prof. G. Boracchi

November 26, 2022

# Contents

# 1 Introduction

This project is part of the exam of *Artificial Neural Networks and Deep Learning* course at Politecnico di Milano, and it corresponds to the first challenge. The aim of the project is to classify species of plants, which are divided into categories according to the species of the plant to which they belong. Being a classification problem, given an image, the goal is to predict the correct class label.

Deep learning [19] is a subfield of machine learning that has been widely used in image classification tasks. By training a convolutional neural network (CNN) [12] on a large dataset of images, the network can learn to identify patterns in the images that are indicative of the various plant species.

The typical deep learning pipeline will be used, building a CNN from scratch and implementing transfer learning and fine tuning with 6 CNN: *ResNet152V2 [7], InceptionV3 [4], InceptionResNetV2 [11], VGG16 [9], DenseNet201 [2], Xception [10], EfficientNetB0 [3].* Tensorflow [17] and in particular Keras [5] framework will be used.

# 2 Data

The dataset consists of 3543 images of plants distributed in 8 different species, with the following distribution: 187, 532, 515, 511, 531, 222, 537, 508. The size of the majority of the images is 96x96. The classes are unbalanced, especially species 1 and 6. This could turn out to be a problem in the training of the models because these classes would be classified as less frequent.

Figure 1: Species 1

Figure 2: Species 2

Figure 3: Species 3

Figure 4: Species 4

Figure 5: Species 5

Figure 6: Species 6

Figure 7: Species 7

Figure 8: Species 8

# 3 Preprocessing

## 3.1 Balancing

In order to solve the unbalanceness of the classes, the initial dataset has been oversampled: the number of images for each class has been set to 537. The classes where the images were lower than this number have been filled with random duplicates of images took from the same class. At the end, all the classes had the same number of images, and the duplicates did not add any bias, being random the procedure of oversampling. An alternative of this approach could have been undersampling, but for the current dataset this option is not efficient because the dataset contains few images.

## 3.2 Splitting

The splitting of the initial dataset has been performed using *split-folders* library [14], obtaining 3 sets: a training set with 70% of the total data, a validation set with 20% of the total data and a test set with 10% of the total data. Further in the development of the models, another splitting has been performed, this time obtaining two sets: a training set with 80% of the initial data and a validation set with 20% of the initial data. The test set was not created and splitted because the model will be tested on a hidden and private test set. All the procedures will use this version of the data, with training set and validation set splits.

## 3.3 Data Augmentation

Data augmentation[18] is a technique used to increase the diversity of the training set by applying random (but realistic) transformations. It has been the first technique used in order to increase the number of images in the training set. The data augmentation strength of mixing up inputs is also something necessary in order to avoid to much overfitting [15], also because the data was previously oversampled to solve the balancing problem. In keras, data augmenation has been performed defining the transformations directly in the Image Data Generator [16], an object that generate batches of tensor image data with real-time data augmentation. These batched are necessary when a lot of images are taken into account and have to be fed in the training phase. Many transformations have been used:

- **Height Shift Range**: specifies the range of values for the vertical shift in pixels;
- **Width Shift Range**: specifies the range of values for the horizontal shift in pixels;
- **Zoom Range**: specifies the range of values for the zoom of the image;
- **Horizontal Flip**: randomly flip inputs horizontally.;
- **Shear Range**: shear intensity (shear angle in counter-clockwise direction in degrees);
- **Fill Mode**: one of "constant", "nearest", "reflect" or "wrap". Default is 'nearest'. Points outside the boundaries of the input are filled according to the given mode;

The main idea is infact building a features extractor that is detached from the fixed position of the elements, but that is more capable to catch the meaning of elements shapes contained in the images, and using augmentation the model is able to be more invariant to those type of transformations. Alternatives techniques have been tried in this phase, such as *CutMix, MixUp* [6] with *KerasCv*, but there were some problems in the environment and therefore they could not be properly tested.

# 4 Models

The models proposed are 7: a network from scratch, *ResNet152V2, InceptionV3, InceptionResNetV2, VGG16, DenseNet201, Xception, EfficientNetB0*. All the model except the network from scratch have been used as feature extractor, using the transfer learning [1] paradigm. Transfer learning is a machine learning technique that allows a pre-trained model, which has already learned patterns and features from one task, to be used as a starting point for a new related task. Instead of training a new model from scratch, transfer learning can improve the performance and speed of learning for the new task by leveraging the knowledge and features learned from the pre-trained model. For example, a pre-trained image recognition model can be fine-tuned for a specific image classification task by retraining the last few layers of the model using a smaller dataset specific to the new task. All of the 6 architectures have been extensively trained on large image datasets such as ImageNet and have achieved high performance on various computer vision tasks, making them popular choices for transfer learning

## 4.1 Network from scratch

A Convolutional Neural Network (CNN) for image classification typically consists of a series of layers that transform the input image into a probability distribution over a set of target classes. The layers typically used in a CNN for image classification are: *Input Layer, Convolutional Layer, Activation Layer, Pooling Layer, Fully Connected Layer, Batch Norm Layer, Dropout layer*. These layers are usually stacked together to form a deep neural network, with the last layer producing the final output of the network. During training, the network learns the weights of the filters and fully connected layers by minimizing a loss function that measures the difference between the predicted and actual class labels of the training examples. Once trained, the network can be used to classify new images by feeding them through the network and taking the class with the highest probability as the predicted label. In order to avoid overfitting, regularization, dropout and batch normalization are used.

## 4.2 ResNet152V2

ResNet152V2 is a deep residual neural network architecture that won the 2015 ImageNet classification challenge. It is a 152-layer network that uses skip connections to address the vanishing gradient problem in very deep networks. It also uses batch normalization and ReLU activation functions, and features residual blocks that allow the network to learn residual mappings.

## 4.3 InceptionV3

InceptionV3 is a convolutional neural network architecture that won the 2015 ImageNet classification challenge. It uses a module called an inception block that consists of multiple convolutional layers with different filter sizes and pooling operations. The outputs of these different layers are concatenated together to form the output of the block. InceptionV3 also uses batch normalization and ReLU activation functions.

### 4.4 InceptionResNetV2

InceptionResNetV2 is a hybrid architecture that combines the InceptionV3 module with residual connections. It is a 164-layer network that uses a combination of convolutional and pooling layers, with residual connections between some of the layers. It also uses batch normalization and ReLU activation functions.

### 4.5 VGG16

VGG16 is a convolutional neural network architecture that was developed by the Visual Geometry Group at Oxford University. It is a 16-layer network that uses small 3x3 filters throughout the network. It also uses max pooling layers and ReLU activation functions.

### 4.6 DenseNet201

DenseNet201 is a deep neural network architecture that uses dense connections between layers. In a dense block, the output of each layer is concatenated with the outputs of all previous layers. This allows the network to reuse features learned by earlier layers and improves gradient flow. DenseNet201 is a 201-layer network that also uses batch normalization and ReLU activation functions.

### 4.7 Xception

Xception is another popular convolutional neural network architecture that was proposed by François Chollet in 2016. It is a deep neural network that uses depthwise separable convolutions, which factorize the standard convolution operation into a depthwise convolution and a pointwise convolution. This allows the network to capture spatial and channel-wise information separately, which reduces the number of parameters and computation required by the network. The Xception architecture is based on the Inception architecture, but replaces the standard convolutional layers in the Inception module with depthwise separable convolutions. It also uses batch normalization and ReLU activation functions, and includes skip connections to improve gradient flow and help the network learn more complex representations.

### 4.8 EfficientNetB0

EfficientNet is a family of neural network architectures that were developed by scaling up the network width, depth, and resolution. The architecture uses a combination of depthwise separable convolutions and squeeze-and-excitation blocks to achieve state-of-the-art performance with fewer parameters and less computation compared to other architectures. EfficientNetB0 is the smallest variant in the EfficientNet family.

# 5 Training

Below, the design choices and the training pipeline of each networks will be described, together with the names of the python notebooks where it is possible to check the implementation details.

## 5.1 Network from scratch

We started with a simple network from scratch.

The problem is classifying leaves (small shaped objects), so we used filters whose size is small (3x3), with padding 'same', since we are also interested in capturing the border of the images. We decided that the number of channels should be low in the first layers so that they detect low-level and local features, and more filters in the next layers in order to capture many complex shapes.

We started from a simple network with 2 convolutional layers, Global Average Pooling (GAP) [13], 2 fully connected layers: we obtained a network with few parameters, and accuracy on the validation set around 0.55.

Increasing the number of convolutional layers led us to a sharp decrease in performances, while keeping this architecture and increasing the depth of the convolutional layers led us to an increase in the number of parameters and also in performances (0.73 on validation), after which we were not able to improve it anymore.

We had the same performance increasing the depth of the 2 convolutional layers from 64 and 128 to 256 and 512, and the depth of the fully connected layer from 512 to 1024 *(MicheleNet1)*, but worse performance when we increased the depth even more, since we had more parameters and we didn't use regularization.

With this network from scratch, we tuned different hyperparameters and we ended up with a configuration that had the best performances: augmentation with zoom, flips and shifts, RMSprop [8] optimizer with a small learning rate, a fully connected part made of GAP and two dense, each with a RELU and a batch norm layer, and the patience set to 20.

## 5.2 ResNet152V2

We decided to do transfer learning using ResNet152V2 as a feature extractor.

As a starting point, we used the best configuration of the fully connected part of the network from scratch, and we trained it keeping the convolutional part untrainable. By increasing the learning rate to 0.01 in this phase we added a slight improvement, while using regularization, reducing the fully connected layer from 2 dense to 1 and introducing a SVM [20] did not improve the network (*RN_v1.0.3*).

With fine tuning, the performances increased: training the last 50% layers improved the validation accuracy to 87%, better than any other proportion of trainable-untrainable layers *(RN_v1.1.2)*. This may be because the first layers are already extracting the low level features in a good way, while the last layers need to be trained to extract high level features related to the specific task or classify leaves. Moreover, in the fine tuning the learning rate was smaller than in the first phase, so that we don't alter the weights of the original network so much, since they were already a good starting point being trained with 'imagenet' data.

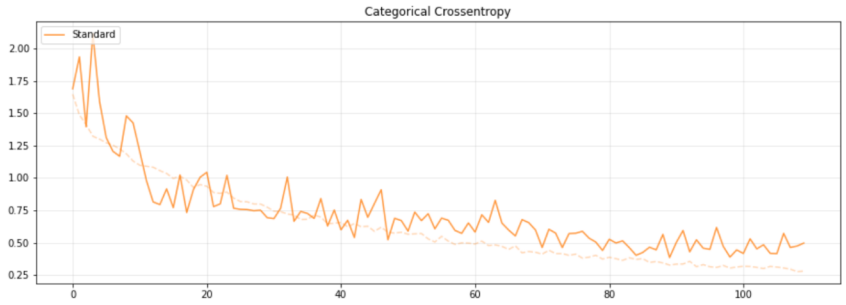


Figure 9: Loss of ResNet152V2

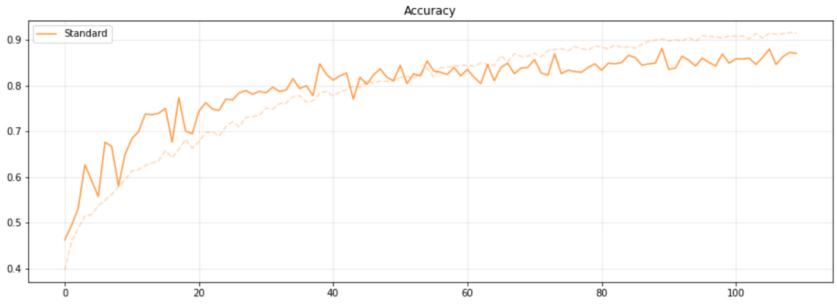


Figure 10: Accuracy of ResNet152V2

## 5.3 InceptionV3

We performed transfer learning using InceptionV3.

We started using the best configuration of the fully connected part used in the network from scratch, plus a regularization to avoid overfitting *(IN_ v1.2.1)*. In the fine tuning phase, we freezed the first 132 layers (almost 50%) that corresponds to 5 inception blocks, and therefore we approached the problem as in the case on ResNet. We reached an accuracy on the validation set of 85% after training on more than 130 epochs *(IN_ v1.2.2)*. Different proportions of untrainable layers led to worse performances.

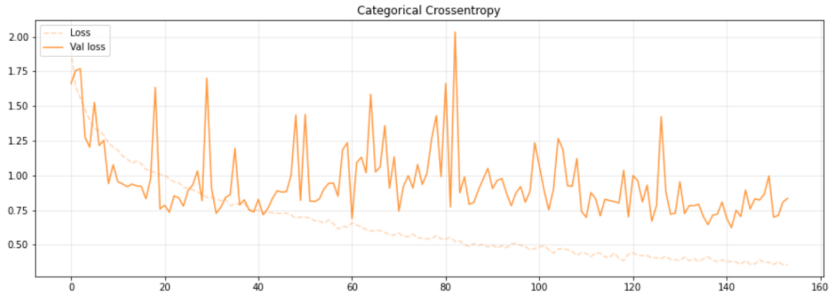


Figure 11: Loss of InceptionV3

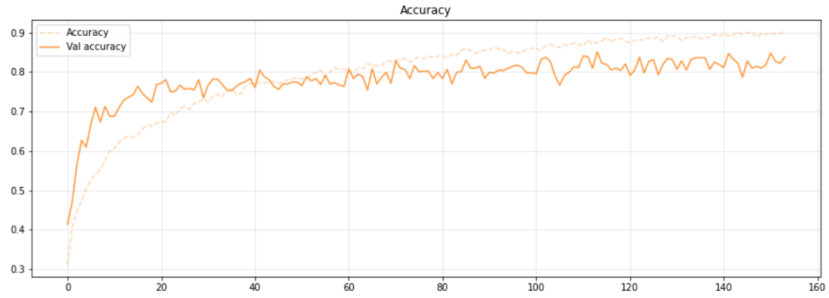


Figure 12: Accuracy of InceptionV3

## 5.4 InceptionResNetV2

We then approached the problem using InceptionResNet as a feature extractor. We started from the configuration seen in the ResNet and Inception models *(IR_ v1.0.2)*.

With the fine tuning, we approached it as we did in ResNet and Inception. We trained the first 30% (since they recognize low level patterns and edges related to the new problem of leaves classification) and the last 30% layers, with an accuracy of 87%. After some trials we had the best performance when we didn't freeze any layers and therefore when we trained all the parameters, reaching an accuracy of 88% after 120 epochs *(IR_ v1.0.4)*.

We also fine tuned this version by removing a dense layer, reaching 90% on the validation set and 77,8% on the test set *(IR_ v1.1.3)*, probably because leaves (not a very hard-shaped object) needed a quite simple classificator. Almost the same performance has been achieved when we retrained it by increasing the dropped neurons in the dropout layer and adding batch normalization layers, with a test accuracy of 76% *IR_ v1.1.4)*.

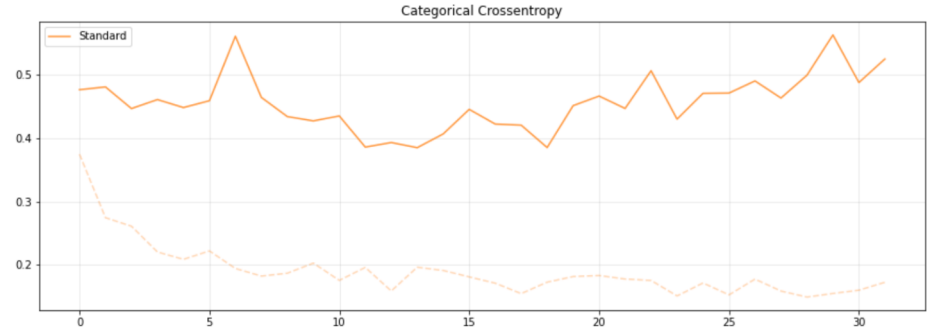


Figure 13: Loss of InceptionResNetV2

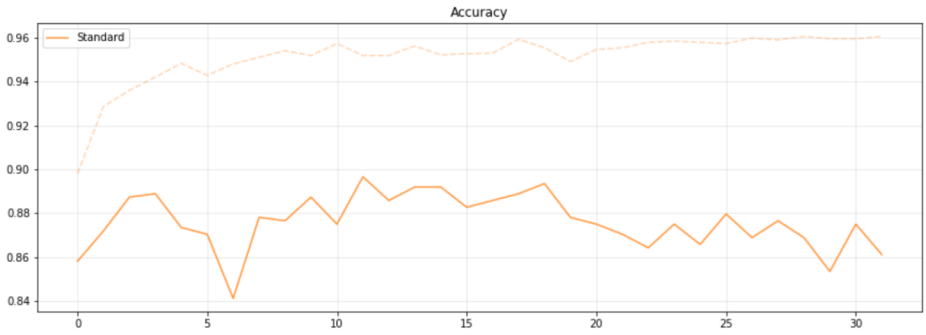


Figure 14: Accuracy of InceptionResNetV2

## 5.5 VGG16

We performed transfer learning with VGG16.

First experiments were done using a rescale of 1/255. Then, later experiments were done without normalization, since it seems to perform better. As first experiment we insert after the VGG16 feature extractor, a Flatten layer, a classifier with 512 neurons and the final output dense layer. We inserted a dropout layer after the VGG16 and the 512 unit classifier. In this first experiment we put the VGG16 non trainable, so the trainable parameter were 2,363,912. We reached a validation accuracy of 55%.

Then we began fine-tuning it. First we set trainable block 5 of VGG16, the last block, freezing the first 10 layers and reaching a validation accuracy of 73%. Then we freezed the first 10 layers, leaving trainable the last 2 blocks (4 and 5) having a total of around 15 million trainable parameters (indicating that most of the parameters are in the deepest layers).

We increased the patience to be more sure that the model was really overfitting and had no more chance to get better. With this configuration we reached a validation accuracy of 83% but the test accuracy was at 80% *(vgg16_fine_tuning_v1)*.

We also tried freezing the first 7 layers, adding another dense layer with 128, GAP + one 256 classifier, reducing augmentation parameters (to reduce overfitting), but the results were unfortunately the same. So we removed the second classifier layer of 128 units and reduced even more the augmentation and the results were immediately better, reaching almost 90% accuracy in validation and 98% in training accuracy.

To try to reduce overfitting we removed dropout after the VGG16 and added regularization (l1=1e-4; l2=1e-2) and did more augmentation but both didn't improve validation.

Then we changed a bit our model, adding a GAP layer, followed by 2 classifier of 512 and 64 units, with dropout and batch normalization compiled with RMSprop optimizer *(vgg16_fine_tuning_v2)*. We got a more oscillating accuracy curve and reached 88% accuracy in validation, but in test accuracy we managed to reach 81%, so a slight improvement with respect to the previous models.

By performing more augmentation and adding regularization we reached a validation accuracy of 91%. Then we tried a new approach: first train for 50 epochs, only the top classifiers dense layers, with a higher learning rate (1e-2) and then unfreeze the last blocks, leaving only the first 10 layers non-trainable, then train with a lower learning rate (1e-4) *(vgg16_fine_tuning_v3)*. With this configuration the accuracy fell down to 89%.

We also tried with the same approach to train first with the first 10 layers frozen (lr=1e-3) and then the first 4 layers frozen (lr=1e-4), but the accuracy went down to 81%. We tried different configuration: 10 layers always frozen with lower learning rate (89% validation), load v3 and froze up to layer 7 (89% validation) *(vgg16_fine_tuning_v4)* more augmentation, dropout and l2 regularization (74% validation; very similar performance in training and validation, indicating a weak model).

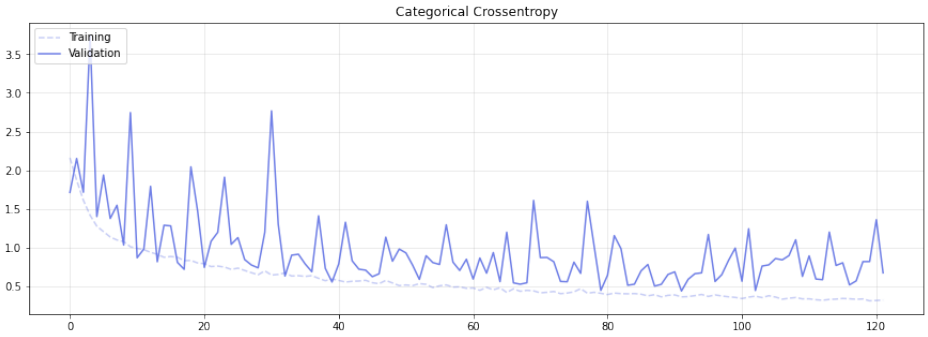


Figure 15: Loss of VGG16

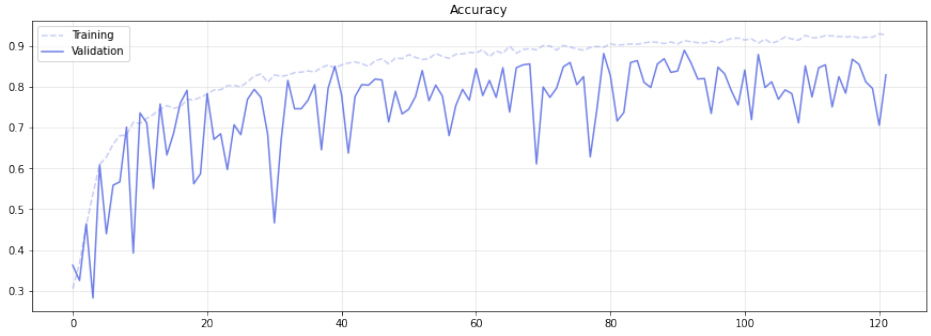


Figure 16: Accuracy of VGG16

## 5.6 DenseNet201

We wanted to experiment if changing the model to fine tune from a low layers one as VGG16 (16 layers, 138.4M parameters) to a significantly deeper one as DenseNet201 (402 layers, 20.2M parameters) would make some differences.

Since the layers here are divided in blocks and there are many, starting from the bottom we unfreezed one block at time looking if performances increased or decreased. Just unfreezing the very last layer produced slower, but not bad results (-18 layers: validation accuracy: 0.6698) and going up (-226 layers: validation accuracy: 0.6816) performance slightly increased but not significantly. Looking to the starting point *(vgg16_fine_tuning_v1)*, we concluded that just changing model and increasing exponentially the number of layers were not the right approach, so we quickly discarded this model.

## 5.7 Xception

First, as before, we trained without fine tuning.

In this case the model seems to not learn, as the validation accuracy oscillates around a low value. So, we tried freezing just the first 6 blocks out of 14 (54/131 layers frozen). The problem was that there was too much variation in data augmentation: as a matter of fact, by reducing data augmentation, we were able to reach 65% accuracy (xception_fine_tuning_v1).

We then tried to create an overfitting net. So we removed data augmentation and worked with a balanced dataset, only with the copies (without augmenting it). To avoid the overfitting we introduced regularization, but the training accuracy saturated at 75%, so we reduced l2 regularization to 1e-5 and in this way on the training we got 95%, but 65-67% in validation. So we reached a good overfitting net; to correct it we increased dropout rate from 0.3 to 0.5 and 0.7 experiencing not much improvement in validation, but no drop in training performance.

## 5.8 EfficientNetB0

At the beginning, we trained the model without fine tuning.

In this case we began using 2 dense layers of 2048 and 512 unit (since with flattening we had a vector of dimension 18432, that seemed a reasonable choice) and Adam as optimizer. We got 94% accuracy in training and 80% in validation, which can be a good overfitting starting point.

We proceeded with a GAP layer and the performance were almost the same, so we decided to continue with GAP and reduce the 2 hidden layers to a single one of 256 neurons. This model had the same performance, indicating that the model doesn't need to become more complex. At this point we freezed all block, except the last 6 and 7 to fine tune. The validation accuracy saturated at 77% so we stopped early the run.

To improve validation, we introduced dropout regularization, and batch normalization obtaining the same performance. We managed to avoid overfitting but on the other hand we incurred in an underfitting model. Adding another classifier of 64 units, unfreezing only the last block, removing dropout after efficient net or removing regularization had very poor improvements.

At this point we used efficientnetB5, which is a more complex model to avoid underfitting. This model without batch, dropout, regularization and with flatten instead of GAP achieved 70% in validation. By adding the augmentation part the model reached 75% and saturated.

We run B0 version (with a GAP and a single 512 classifier) in 2 phases like in VGG16 experiments: 6a and 5a *(efficientnet_fine_tuning_v2)* or 6a and all net *(efficientnet_fine_tuning_v3)*, getting no improvements. Also B5 reached 50% in validation *(efficientnet_fine_tuning_v4)*.

At this point we removed the classifier, and used only a GAP and the output layer together with the efficientnetb5 *(efficientnet_fine_tuning_v5)*. Trained in 2 phases (first all and then from block 5a) we reached 90% in validation, and very high results in training. However, the 2 phases seemed useless, since in the second phase there was no improvement.

At this point we added more augmentation with just 1 phase and B0, B3 and B5 *(efficientnet_fine_tuning_v6)* networks. All of them reached 90% in validation accuracy with B0 slightly below.

At this point we implemented an ensemble method, since we had good overfitting network: a bagging with 10 B0 models as before *(efficientnet_fine_tuning_bagging)*. During training the models overfitted very fast, keeping very bad validation performance. This was probably due to many repeated samples. Testing on the online test set we got an out of time error from the server, so we decided to not to elaborate more this strategy, since there was no way to test it.

For the final test we loaded the previous B3 model with only the GAP, froze it and add a 128 classifier (the only part trainable). We got a little improvement, reaching 91%. By performing more augmentation and regularization, we reached the same performance but obtained around 96% in training instead of almost 100% *(efficientnet_fine_tuning_v7)*. We noticed that the slope of the loss was still going down when stopped, so we decided to load the last model and stop when the loss was at the minimum, but this didn't change the performance. This last model was the one providing 82% on the online test, and the best model we were able to create.

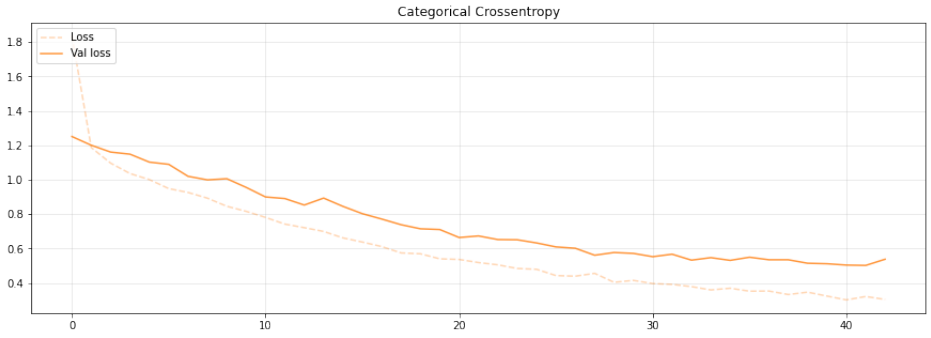


Figure 17: Loss of EfficinentNetB0

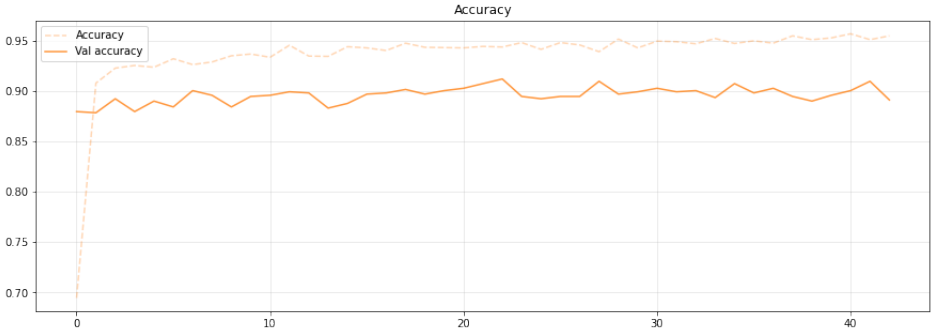


Figure 18: Accuracy of EfficientNetB0

# 6 Results

Implementing a network from scratch helped to explore the problem and understand what are the most suitable structures for a network based on this specific problem of classification. The performances where not low, since the accuracy on the validation reached 78%.

Transfer learning was the tool that obtained the best result, since very powerful model has been used to extract in an optimized and efficient way the features form the images, being already trained with a hige corpus of images.

All the models that ahve been used as feature extractor plus a fully connected part related to the problem performed quite well, having accuracy above 85%. EfficientNet was the best among the other, with high performances, around 91% in the accuracy of the validation set, and 82% on the hidden test set.

# 7 Conclusions

Starting from scratch, it was hard to make a good network, due to the limited number of images. Therefore, using transfer learning was a good choice in terms of training time, as well as initialization, since those networks already had optimized weights. However, very powerful networks are not always the way to go because they really depend on the problem faced. With these networks it is quite easy to reach a good result, but very hard to fine tune them to achieve an even higher performance.

For the team work organization, in the data preparation phase, we discussed all together the best strategies to follow, while in the development each member focused on some models, giving her results to the others and, when necessary, defining all together the next steps to follow. The final work has been revisioned by all the members during the last days of the competition.

# References


[1] Jason Brownlee. A gentle introduction to transfer learning for deep learning. https://machinelearningmastery.com/transfer-learning-for-deep-learning/.

[2] Keras. Densenet201. https://keras.io/api/applications/densenet/#densenet201-function.

[3] Keras. Efficientnetb0. https://keras.io/api/applications/efficientnet/#efficientnetb0-function.

[4] Keras. Inceptionv3. https://keras.io/api/applications/inceptionv3/.

[5] Keras. Keras. https://keras.io/.

[6] Keras. Random augmentations. https://keras.io/guides/keras_cv/cut_mix_mix_up_and_rand_augment.

[7] Keras. Resnet152v2. https://keras.io/api/applications/resnet/#resnet152v2-function.

[8] Keras. Rmsprop. https://keras.io/api/optimizers/rmsprop/.

[9] Keras. Vgg16. https://keras.io/api/applications/vgg/#vgg16-function.

[10] Keras. Xception. https://keras.io/api/applications/xception/.

[11] Kerasl. Inceptionresnetv2. https://keras.io/api/applications/inceptionresnetv2/.

[12] Manav Mandal. Introduction to convolutional neural networks (cnn). https://www.analyticsvidhya.com/blog/2021/05/convolutional-neural-networks-cnn/.

[13] Oreolorun Olu-Ipinlaye. Global pooling in convolutional neural networks. https://blog.paperspace.com/global-pooling-in-convolutional-neural-networks/.

[14] Pypi. Split folders. https://pypi.org/project/split-folders/.

[15] Rutger Ruizendaal. Deep learning 3: More on cnns handling overfitting. https://towardsdatascience.com/deep-learning-3-more-on-cnns-handling-overfitting-2bd5d99abe5d.

[16] Tensorflow. Image data generator. https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator.

[17] Tensorflow. Tensorflow. https://www.tensorflow.org/.

[18] Wikipedia. Data augmentation. https://en.wikipedia.org/wiki/Data_augmentation.

[19] Wikipedia. Deep learning. https://en.wikipedia.org/wiki/Deep_learning.

[20] Wikipedia. Support vector machine. https://en.wikipedia.org/wiki/Support_vector_machine.