UNIVERSITÀ DI PISA

Artificial Intelligence and Data Engineering

Computational Intelligence and Deep Learning

Artist Recognition

Project Documentation

TEAM MEMBERS:
Fabiano Pilia
Elisa De Filomeno

Academic Year: 2022/2023

Contents

1 Introduction

This project aims to study the application of AI algorithms in the context of artists recognition. The problem of identifying the artist based on an image of a painting is very challenging, the same artist may have a high variability in artistic styles and different artists may have similar styles. The objective is to develop different Convolutional Neural Networks (CNNs) from scratch and using pre-trained CNN models to solve this problem. In this report, we will present our findings and observations from these experiments.

1.1 State-of-Art artist recognition

For many decades researches tried to apply image processing techniques for recognizing artists through the analysis of paintings. However, only in recent years artificial intelligence led to a rapid evolution in this field. In the past there was also the problem that the necessary data for research wasn't widely available and only in 2008 there has been one of the first attempts to make available a high resolution and richer data set. The Van Gogh and Kröller-Müller Museums in The Netherlands agreed to make available a data set of 101 high-resolution gray-scale scans for multiple research groups. Of the 101 paintings mostly have been attributed to van Gogh. Experts used image processing techniques (in this case wavelet transorfmations) to extract distinctive features of paintings, such as edges, contrast, orientation, and texture. These features were then used to create artist recognition models, based on the 2-D HMM (Hidden Markov Model) statistical model and SVMs, reaching an accuracy of 67% in classifiyng van Gogh paintings. In 2013 several classifiers, like multi-layer perceptron (MLP), sequential minimal optimization for support vector machine (SMO), Naive Bayes classifier, random forest and AdaBoost, were tested on a collection of 500 digitized images of paintings from 20 different artists, obtained from various Internet sources, achieving an overall classification accuracy of 75%. Even in this case they used algorithms based on image features on color or textural features. However in recent years AI algorithms based on machine learning techniques managed to achieve even better results! These algorthims, such as deep learning, are able to automatically recognize the distinctive features of paintings. For example in 2022 Tanni Dhoom used CNNs from scratch to determine the artist of a painting out of 6 well-known Bangladeshi artists reaching an accuracy of 89%. However, it is important to note that the accuracies obtained very depend on the specific dataset used, images sizes and quality, pre-processing techniques used and the number of artists included.

References:

- C. R. Johnson et al., "Image processing for artist identification," in IEEE Signal Processing Magazine, vol. 25, no. 4, pp. 37-48, July 2008, doi: 10.1109/MSP.2008.923513. Link
- E. Cetinic and S. Grgic, "Automated painter recognition based on image feature extraction," Proceedings ELMAR-2013, Zadar, Croatia, 2013, pp. 19-22. Link
- Dhoom, Tanni & Sayeed, Dr. (2022). Artwork classification and recognition system based on Convolutional Neural Network. 5. Link

1.2 Dataset

The dataset for this project is taken from Kaggle[1]. It is composed by a collection of paintings belonging to 50 different artists. The file downloaded from Kaggle has three contents:

- **artists.csv**: a csv file containing information about each artist
- **images.zip**: collection of images (full size), divided in folders and sequentially numbered
- **resized.zip**: same collection, but images have been resized and extracted from folder structure

The *artists.csv* file contains information such as name of the artist, genre, Wikipedia page, number of paintings in the dataset and so on. In the following image, we can see an example of the first rows of the file.

	id	name	years	genre	nationality	bio	wikipedia	paintings
0	0	Amedeo Modigliani	1884 - 1920	Expressionism	Italian	Amedeo Clemente Modigliani (Italian pronunciat...	http://en.wikipedia.org/wiki/Amedeo_Modigliani	193
1	1	Vasiliy Kandinskiy	1866 - 1944	Expressionism,Abstractionism	Russian	Wassily Wassilyevich Kandinsky (Russian: Васи...	http://en.wikipedia.org/wiki/Wassily_Kandinsky	88
2	2	Diego Rivera	1886 - 1957	Social Realism,Muralism	Mexican	Diego María de la Concepción Juan Nepomuceno E...	http://en.wikipedia.org/wiki/Diego_Rivera	70
3	3	Claude Monet	1840 - 1926	Impressionism	French	Oscar-Claude Monet (; French: [klod mɔnɛ]; 14 ...	http://en.wikipedia.org/wiki/Claude_Monet	73
4	4	Rene Magritte	1898 - 1967	Surrealism,Impressionism	Belgian	René François Ghislain Magritte (French: [ʁəne...	http://en.wikipedia.org/wiki/René_Magritte	194
5	5	Salvador Dali	1904 - 1989	Surrealism	Spanish	Salvador Domingo Felipe Jacinto Dalí i Domènec...	http://en.wikipedia.org/wiki/Salvador_Dalí	139
6	6	Edouard Manet	1832 - 1883	Realism,Impressionism	French	Édouard Manet (US: ; UK: ; French: [edwaʁ manɛ...	http://en.wikipedia.org/wiki/Édouard_Manet	90

Figure 1: First 7 rows of the file *artists.csv*.

For us there are 2 important attributes: the *name* that will be the classes of the project and *paintings* that will be used in the computation of the weights. The others are not useful information for our purposes.

As anticipated the second file is a zip file containing the folders with the paintings. The structure of folders inside the file reflects the different classes of this dataset, in fact, their names are the names of the artists of whom the paintings images are contained inside the folder. The tree of directories is shown in the figure 2.

The third file has the same format of the second one, but is not considered in our project.

[1]https://www.kaggle.com/ikarus777/best-artworks-of-all-time

Figure 2: Structure of the directories inside the *images.zip* file.

2 Dataset preprocessing

In this section are present all the manipulation done in order to adapt the dataset for this project.

2.1 Download the dataset

Initially we've downloaded the dataset directly form Kaggle, after connecting to it, using the following command:

```
#Download the dataset
kaggle datasets download -d ikarus777/best-artworks-of-all-time

#Unzip the dataset
unzip best-artworks-of-all-time.zip
```

2.2 Correction of the classes names'

Analyzing the dataset we have seen that are present two folders referred to the artist Albrecht Dürer. From the comparison of the content we've found that the two folders are duplicates. To solve this problem the first folder has been dropped using the **shutil** module of python:

```
[ ]  shutil.rmtree('images/images/Albrecht_DuTXa┤Krer')
```

The name was composed by non ASCII characters. Another problem related to this artist was the different name of its folder in the images.zip file respect to its name in the artists.csv file, since in the latter we have information about the classes we have decided to change the name of the folder to **Albrecht Dürer**. The next step was to normalize the names of the folders of the other artists in order to have the same name as it was in the artists.csv file. At this point we've a dataset without duplicates and with a coherent classes names.

2.3 Dataset reduction

For our project we decided not to use all the artists present in the kaggle dataset.

We counted the number of paintings and classes in the dataset considering different numbers of paintings per artist. In the table below we report the numbers obtained.

N paintings per artist	number of classes	tot. number of paintings
any	50	8446
> 100	30	7091
> 150	18	5576

We decided to use only the artists with more than 150 paintings so that our model has enough training samples for each class. In this way we reduced the number of classes from 50 to 18 and the total number of paintings from 8446 to 5576. Then we saved the information about the reduced dataset in *artists.csv*.

	name	genre	paintings
0	Amedeo Modigliani	Expressionism	193
1	Rene Magritte	Surrealism,Impressionism	194
2	Vincent van Gogh	Post-Impressionism	877
3	Mikhail Vrubel	Symbolism	171
4	Pablo Picasso	Cubism	439
5	Pierre-Auguste Renoir	Impressionism	336
6	Francisco Goya	Romanticism	291
7	Albrecht Dürer	Northern Renaissance	328
8	Alfred Sisley	Impressionism	259
9	Marc Chagall	Primitivism	239
10	Sandro Botticelli	Early Renaissance	164
11	Henri Matisse	Impressionism,Post-Impressionism	186
12	Edgar Degas	Impressionism	702
13	Rembrandt	Baroque	262
14	Titian	High Renaissance,Mannerism	255
15	Paul Klee	Expressionism,Abstractionism,Surrealism	188
16	Andy Warhol	Pop Art	181
17	Paul Gauguin	Symbolism,Post-Impressionism	311

Figure 3: modified file *artists.csv*.

In the following plot we show the number of paintings per artist in the reduced dataset:

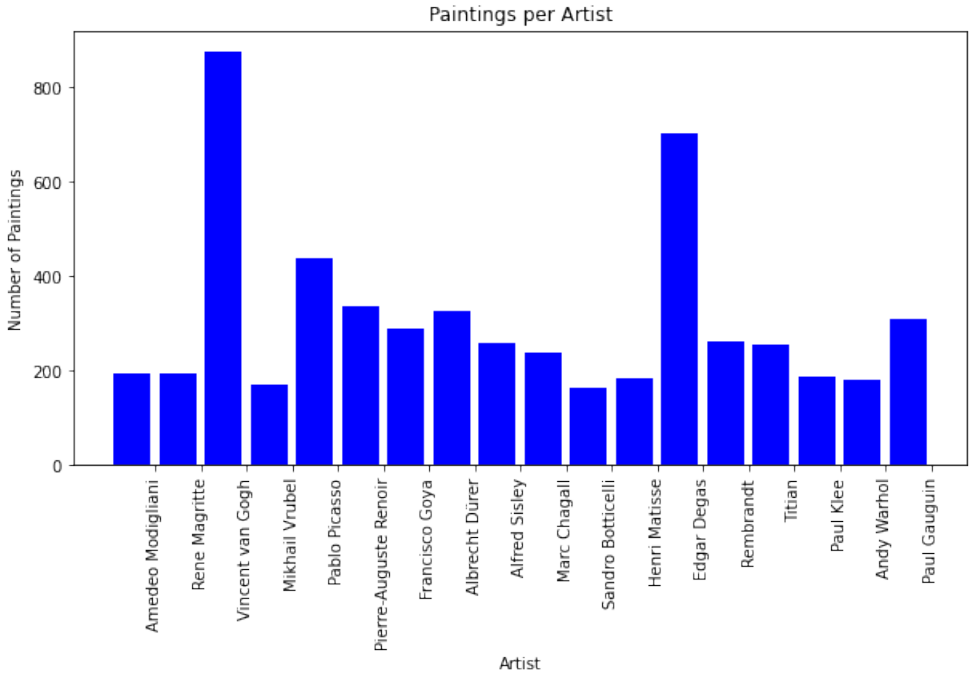


Figure 4: reduced dataset distribution

2.4 Building training and testing set

In this phase we've split the dataset in two parts: 80% of the dataset for the training/-validation set and remaining 20% for the testing set. In order to preserve the hierarchy of the folders we've used an imported function called *splitfolders* that, given the ratio of split, it will split each folders content recursively following the ratio.

```
import splitfolders

splitfolders.ratio("images/images150", output="images/output150",
    seed=1337, ratio=(.8, .2), group_prefix=None, move=False)
```

To check if the split was actually correct we've made the intersection between the content of the training set folders and the content of the test set folders, we obtained an empty set which confirmed the correctness of the split.

The result of the split is the following:

- training/validation set: 4452 paintings
- test set: 1112 paintings

2.4.1 Training set distribution

In the following plot we can see the class distribution of our training/validation dataset:

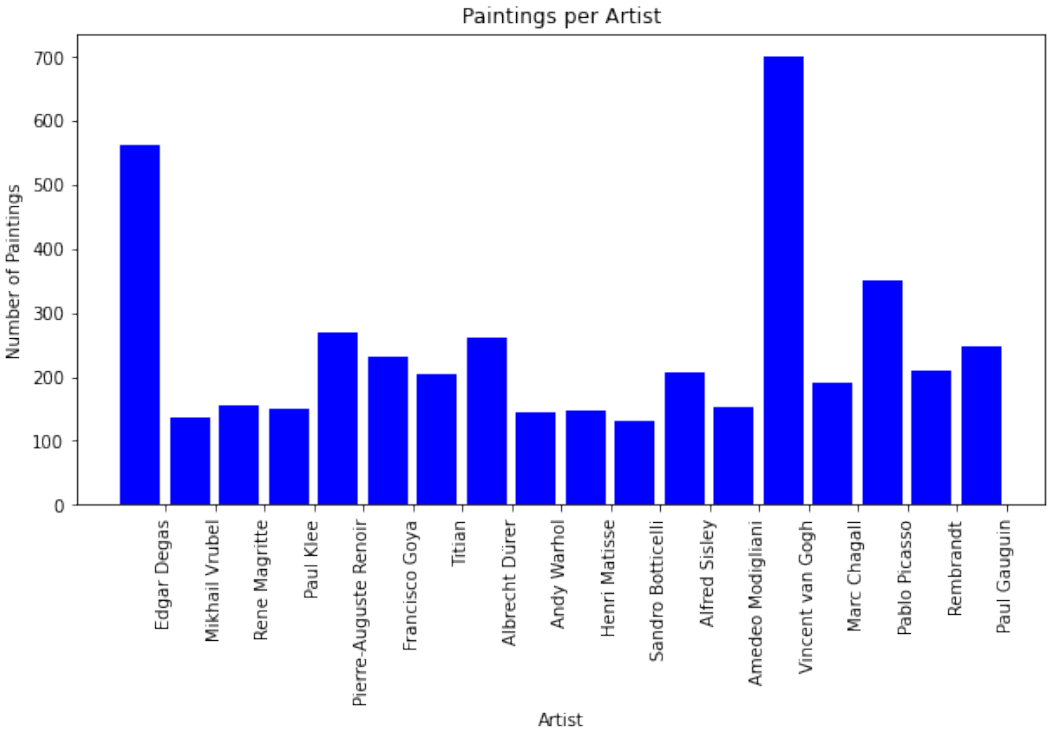


Figure 5: Paintings distribution per artist

As we said before in total in the training/validation set we have 4452 paintings. The smallest class (*Sandro Botticelli*) has 131 samples, the highest (*Vincent van Gogh*) has 701 samples. As a rule of thumbs, if the dataset has a difference (majority class - minority class) of greater than 65% with respect to the majority class is considered unbalanced. In our case this percentage is ∼81.3%, so we can consider our dataset unbalanced.

We used two alternatives approaches to handle the unbalanced training set:

- **Data Augmentation to balance the training set**: this technique let us to increase the diversity of the training set by applying random transformations, such as image rotation, zoom and flip. We've used this method in order to create artificial samples used to balance the number of samples among the classes. In addition to this method we've reduced the majority classes' samples.
- **Class weights with an unbalanced training set**: *class weights* is a parameter that will be passed in the fit() of our model, it maps each class to a weight.

2.5 Assign weights to classes

Since the classes were imbalanced we wanted to use weights, we wanted to assign higher weight to the classes with less data in order to be learned more by the network. We've tried two different approaches, the first using the sklearn library and the second implemented by ourselves.

The first method consists in invoking the function *class_weight.compute_class_weight()*, that assumes that the dataset is a set of samples in the form of (image, class), in fact the expected parameters requires the dataset in such a format. Unfortunately our dataset is

a tree of directories in which each one is a class, so this function is not compatible with our dataset structure, therefore we need to find another way to compute the weights of the classes.

To solve the previous problem we've decide to compute for each class its weight using the following formula:

$$\frac{T}{C * N_i}$$

T	total number of samples
C	number of classes
N_i	number of samples in the class i

This formula is the same used in the *sklearn* function, but using our code it was possible to cope with the structure of our dataset.
The weights can be seen in the figure 6, ordered in increasing order.

	name	genre	paintings	weight
2	Vincent van Gogh	Post-Impressionism	877	0.353224
12	Edgar Degas	Impressionism	702	0.441279
4	Pablo Picasso	Cubism	439	0.705644
5	Pierre-Auguste Renoir	Impressionism	336	0.921958
7	Albrecht Dürer	Northern Renaissance	328	0.944444
17	Paul Gauguin	Symbolism,Post-Impressionism	311	0.996070
6	Francisco Goya	Romanticism	291	1.064528
13	Rembrandt	Baroque	262	1.182358
8	Alfred Sisley	Impressionism	259	1.196053
14	Titian	High Renaissance,Mannerism	255	1.214815
9	Marc Chagall	Primitivism	239	1.296141
1	Rene Magritte	Surrealism,Impressionism	194	1.596793
0	Amedeo Modigliani	Expressionism	193	1.605066
15	Paul Klee	Expressionism,Abstractionism,Surrealism	188	1.647754
11	Henri Matisse	Impressionism,Post-Impressionism	186	1.665472
16	Andy Warhol	Pop Art	181	1.711479
3	Mikhail Vrubel	Symbolism	171	1.811566
10	Sandro Botticelli	Early Renaissance	164	1.888889

Figure 6: The result of the weighting process

As we can see in figure 6, the minimum weight is associated to the artist with the highest number of paintings, while the maximum weight is associated to the artists with the lowest number of paintings. This is what we've wanted since for the training process we must pay more attention to the classes with the less number of samples.
The weights will be used during the training of the model that uses the unbalanced dataset.

3 Creation of the balanced training set

As we said before another approach alternative to using class weights is to balance the training set. We modified our training set in order to have 450 samples for each class. We proceeded in this way:

- if the classes had more than 450 samples we removed the excessive samples from the training set and added them to testing set to still use them.
- if the classes had less than 450 samples we used the *Data Augmentation* technique to increase the number of samples in this classes.

The value *450* was obtained after different tries: with a lower number of samples no significant improvement was observed during the training of the model, while with more samples the training of the model was too slow due to the limitation of the resources provided by the Google Colab framework.

3.1 Data Augmentation

We decided to apply some transformations that yield believable-looking paintings, in order to still recognize the artist in a plausible way. The model will train with an higher number of training samples, which helps in the generalization process. The code used is:

```
from tensorflow.keras import layers

data_augmentation = keras.Sequential([
  layers.RandomFlip("horizontal"),
  layers.RandomRotation(0.1),
  layers.RandomZoom(0.2)
])
```

RandomFlip("horizontal") applies horizontal flipping to a random 50% of the images.

RandomRotation(0.1) rotates the input images by a random value in the range[–10%, +10%] (fraction of full circle [-36°, 36°]).

RandomZoom(0.2) zooms in or out of the image by a random factor in the range [-20%, +20%].

In the following picture we can see an example of an image to which is applied *Data Augmentation*:

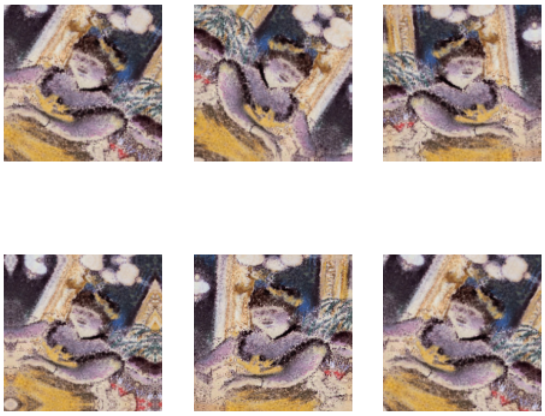


Figure 7: An example of Data Augmentation applied to an image

3.2 Balanced training set distribution

At the end of the process we obtained a balanced training set, as we can see from the following plot all the classes in the training set are now having 450 samples.

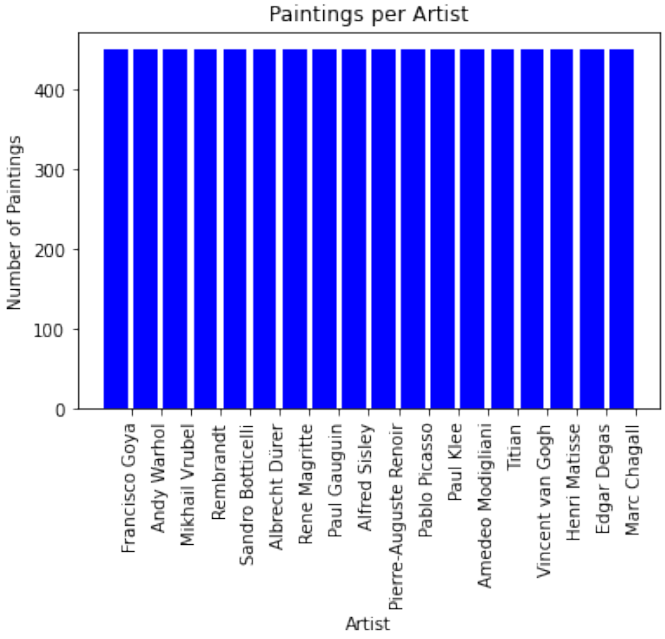


Figure 8: Balanced training set distribution

The total number of samples in the training set are now 8100, while the new number of samples in the testing set are 1474. Of course this test set will be used only for the models that train with the balanced training set, since the training and test set mustn't share any sample.

4 CNN from Scratch

In this section we will report how we developed a CNN from scratch that recognizes the artist starting from the image of a painting. We will report the results obtained from the test of different models and some considerations.

4.1 Preparing data

In this phase we read the png pictures from the datasets, resized them to 180x180 (*IMAGE_HEIGHT* X *IMAGE_WIDTH*), decoded them to RGB grids of pixels and then formatted as floating-point tensors, in this way they can be fed into our machine-learning models.

To prepare the data we used the Keras util *image_dataset_from_directory* which returns a *tf.data.Dataset* that yields batches of images from the directories structure of our training and testing directories. We decided to use a batch size of 64 images.

Moreover with this function we were able to split in two the images in the training directory, 90% of the images will be used for the training set and the remaining 10% of images for the validation set. The test set is composed by all the images contained in the test directory.

The code used is:

```
from tensorflow.keras.utils import image_dataset_from_directory

train_dataset = image_dataset_from_directory(
    balanced_train_dir,
    labels='inferred',
    label_mode='categorical',
    class_names=None,
    color_mode='rgb',
    image_size=(IMAGE_HEIGHT, IMAGE_WIDTH),
    batch_size=BATCH_SIZE,
    validation_split=0.1,
    subset="training",
    shuffle=True,
    seed=1024)

validation_dataset = image_dataset_from_directory(
    balanced_train_dir,
    labels='inferred',
    label_mode='categorical',
    class_names=None,
    color_mode='rgb',
    image_size=(IMAGE_HEIGHT, IMAGE_WIDTH),
    batch_size=BATCH_SIZE,
    validation_split=0.1,
    subset="validation",
    shuffle=True,
```

```
    seed=1024)

test_dataset = image_dataset_from_directory(
    test_dir,
    labels='inferred',
    label_mode='categorical',
    class_names=None,
    color_mode='rgb',
    image_size=(IMAGE_HEIGHT, IMAGE_WIDTH),
    batch_size=BATCH_SIZE)
```

In the above code *labels='inferred'* means that labels are generated from the directory structure (the names of the directories of the images).

While *label_mode = 'categorical'* means that the labels are encoded as a categorical vector, in fact we will use a categorical_crossentropy loss function.

Another preparation of our data consisted in rescaling the float values of the input tensors of our convnets from the [0, 255] range to the [0, 1] range, since small values are in general preferred to neural networks. To do this we added a layer to all our models, the code used for this layer is:

```
x = layers.Rescaling(1./255)(inputs)
```

4.2 Choosing Last-layer activation and Loss function

In this phase we chose the Last-layer activation and Loss function to use to build our models. The choice depended on the type of our problem in this way:

Problem type: multiclass, single-label classification

↓

- **Last-layer activation**: *softmax*
- **Loss function**: *categorical_crossentropy*

4.3 Choosing the starting model

At this point we've decided the starting model on which we will test the two datasets. Initially we've defined the input of the model, the shape's tuple is formed by *(image_height, image_width, image_channels)*, since we're working with images with an height of 180, a width of 180 and 3 channels since we're using RGB images. As anticipated as first layer we've put a rescaling layer that scales the tensors' values to numbers in the [0,1] interval, more suitable for neural network.

```
inputs = keras.Input(shape=(180, 180, 3))
x = layers.Rescaling(1./255)(inputs)
```

We decided to use as the first convnet a stack of alternated 5 Conv2D and 4 MaxPooling2D layers. We've selected a number of channels that are powers of two in an increasing order, the last two pairs of conv2D and MaxPoolingD2 will be used with the same number of channels (256).

We report the code of the first convnet used:

```
x = layers.Conv2D(filters=32, kernel_size=3, strides=1 , activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=64, kernel_size=3, strides=1, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=128, kernel_size=3, strides=1, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=256, kernel_size=3, strides=1, activation="relu")(x)
x = layers.MaxPooling2D(pool_size=2)(x)
x = layers.Conv2D(filters=256, kernel_size=3, strides=1, activation="relu")(x)
```

Conv2D layers arguments used are:

- **filters** = 32, 64, 128, 256, 256
- **kernel_size** = 3
- **strides** = 1, meaning no strides
- **padding** = *valid*, default value, meaning no padding
- **activation** = *relu*

MaxPooling2D layers arguments used are:

- **pool_size** = 2, in order to downsample the feature maps by a factor of 2.

The last step was to use a layer of densely connected classifier network, to do this we've firstly flattened the 3D tensor in order to obtain a 1D vector that is the input expected by the densely connected layer.
Since we're doing a 18-way classification, where 18 is the number of classes, we've passed that number as a parameter for the densely connected layer and as activation function we've decided to use the *softmax*, since it's the most suitable for multi-class classification with a single label for each class.

The code for the last step is:

```
x = layers.Flatten()(x)
outputs = layers.Dense(18, activation="softmax")(x)
```

Finally we chose *Adam* to be our optimizer with a learning rate of 10^{-3}. Let's recap:

- **Last-layer activation**: *softmax*
- **Loss function**: *categorical_crossentropy*
- **Optimization configuration**: *Adam* with learning rate 10^{-3}

The figure 12 contains the model's summary, in which it's highlighted the shape of the sequence of input and outputs. As we can see the flatten layer transform a tensor with shape (7,7,256) into a 1D vector with a shape of (12544,), that will be the input of the dense layer.

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 180, 180, 3)]     0

 rescaling (Rescaling)       (None, 180, 180, 3)       0

 conv2d (Conv2D)             (None, 178, 178, 32)      896

 max_pooling2d (MaxPooling2D  (None, 89, 89, 32)       0
 )

 conv2d_1 (Conv2D)           (None, 87, 87, 64)        18496

 max_pooling2d_1 (MaxPooling  (None, 43, 43, 64)       0
 2D)

 conv2d_2 (Conv2D)           (None, 41, 41, 128)       73856

 max_pooling2d_2 (MaxPooling  (None, 20, 20, 128)      0
 2D)

 conv2d_3 (Conv2D)           (None, 18, 18, 256)       295168

 max_pooling2d_3 (MaxPooling  (None, 9, 9, 256)        0
 2D)

 conv2d_4 (Conv2D)           (None, 7, 7, 256)         590080

 flatten (Flatten)           (None, 12544)             0

 dense (Dense)               (None, 18)                225810

=================================================================
Total params: 1,204,306
Trainable params: 1,204,306
Non-trainable params: 0
_________________________________________________________________
```

Figure 9: First model used

4.4 Starting model using the unbalanced training set

The first test that we have performed using the starting model consists of its training with the unbalanced data set, leveraging on the use of weights. As we mentioned earlier, the weights are calculated during the pre-processing of the data set and stored in the *artists.csv* files. So, the first step was to retrieve the information about the artists and store them in a data frame.
The second step was the creation of a dictionary, since the weights accepted by the *model.fit()* call must be a dict with an integer associated with a class as a key and the weight of the class as value. Then we have built the starting network according to the choices that we have previously mentioned.
The model summary can be seen in figure 12.

The model was compiled using the *Adam* optimizer with a learning rate of 10^{-3}, the *categorical_crossentropy* as loss function and the accuracy as evaluation metric.
We've decided to use the following parameters to load the dataset:

- **Batch size:** 64;
- **Image height:** 180;
- **Image width:** 180.

In this way after 64 samples the model will be updated, and the input samples will be resized with a shape of (180, 180, 3). The following step was to start the training of the model. We've decided to use 25 epochs and, as anticipated, the class weights to pay more attention during the training to the classes with less samples.
The results are presented in the table 1, as we can see we reach high values of accuracy on the training set while we've low values on both the validation and the test set. These values suggest us that we've a case of overfitting.

Starting CNN model		
Number of Epochs: 25		
	Loss	**Accuracy**
Train	0.1259	0.9394
Validation	4.5217	0.4236
Test	4.9067	0.4245

Table 1: Training results with the starting model on the unbalanced dataset

More information can be seen in the following graphs, in the figure 10 we can see the characteristic plot of overfitting, in fact the training accuracy and the validation accuracy are increasing together to the 7th/8th epoch, starting from with the first continues to grow linearly reaching 95% of accuracy, while the validation accuracy stalls around 40%. In the figure 11 we can see the plot of the losses; they follow the same behaviour of the accuracies, so the training loss converges to low values of loss (0.1), while the validation loss reaches values that stall around 4.5.

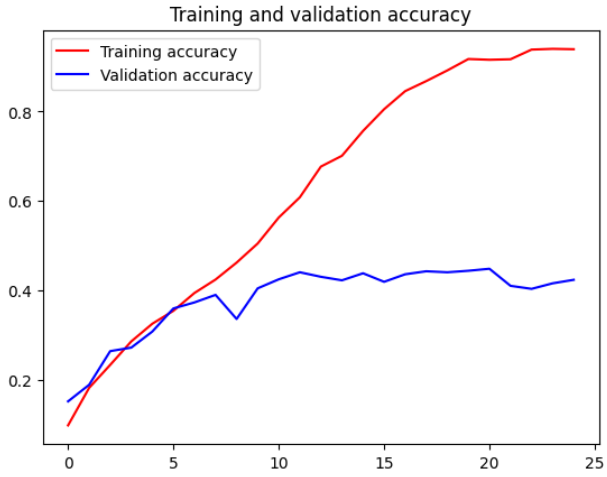


Figure 10: Plot of the training accuracy and validation accuracy during the epochs

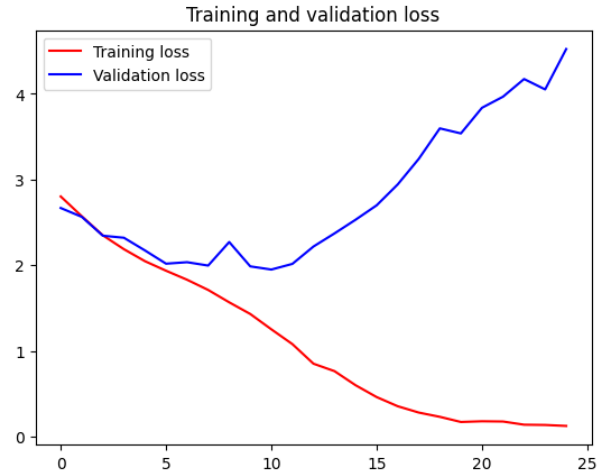


Figure 11: Plot of the training loss and validation loss during the epochs

The values of loss and accuracy on the test set are the ones that we were expecting, in fact they are similar to the values observed in the validation set. this behavior confirms the fact that our starting model is suffering from overfitting.
In order to mitigate the problem of overfitting we can use different techniques:

- Add more data;
- Use data augmentation;
- Use architectures that generalize well;
- Add dropout layers;
- Add L1/L2 regularization;
- Reduce architecture complexity.

In chapter 3 we have exposed the method used to create the balanced dataset: it basically consists in adding more data to the training set and create the artifical samples exploiting

the technique of the data augmentation. So the balanced dataset was created to reduce the overfitting observed using the unbalanced training set.

4.5 Starting model using the balanced training set

The second test was the same done before, but using the balanced dataset, in this case the class weights were not needed.
The architecture of the network is exactly the same, but we've a richer dataset than before, that includes the samples used before. The increment of the dataset is also aimed to fight the overfitting, since we use more samples that are created using data augmentation techniques.
The results can be seen in the following figure:

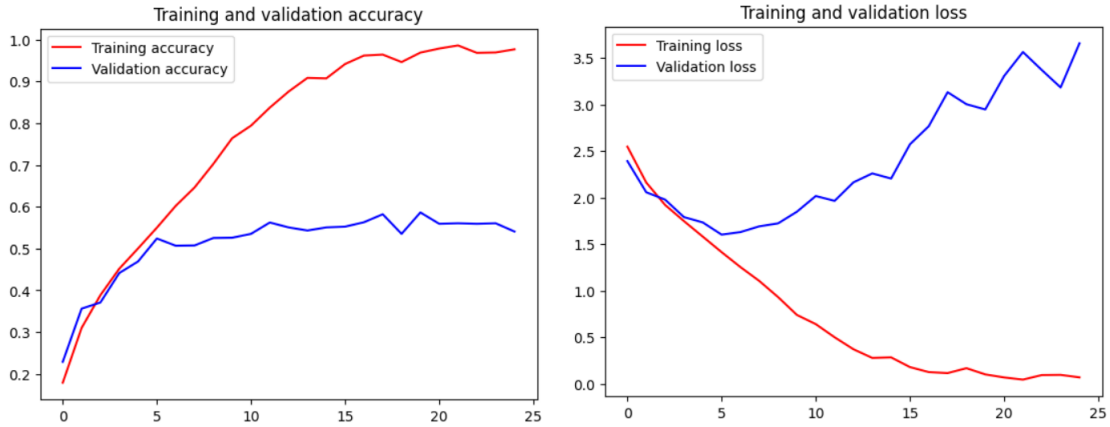


Figure 12: Training and validation accuracy and loss plots of the first model

We can see that starting from the epoch 5 the we encounter overfitting. The model stops to generalize and this can be seen from the stall that affects the validation accuracy plot around the value of 0.54, while the training accuracy continues to grow.
In the following table we can see the values reached at the last epoch (25).

Standard CNN		
Number of Epochs:	25	
	Loss	**Accuracy**
Train	0.0685	0.9769
Validation	3.6572	0.5407
Test	5.3315	0.4505

Table 2: Results achieved by the first model

In conclusion also in this case the model overfits the data. Starting from the results obtained in the previous two tests, we'll compare the two trained models and select the dataset that we'll use to train the future models.

4.6 Choosing the dataset

The following step was to select between the two datasets. To compare the two training results we've plotted their graphs, one against the other. In the figure 13 we can see that the training accuracy plot of the balanced training is always above the plot of the unbalanced training, while the training loss of the balanced training is always under the unbalanced training curve. We can observe significant improvements using the balanced dataset.

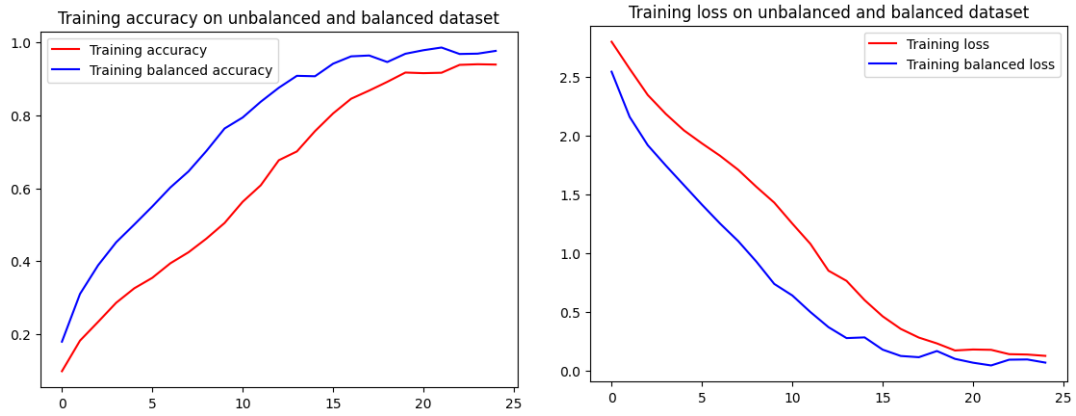


Figure 13: Plot of the training accuracy/loss of the unbalanced dataset vs balanced dataset

The same situation is observed in the validation accuracies and losses, the training on the balanced dataset provides better results.

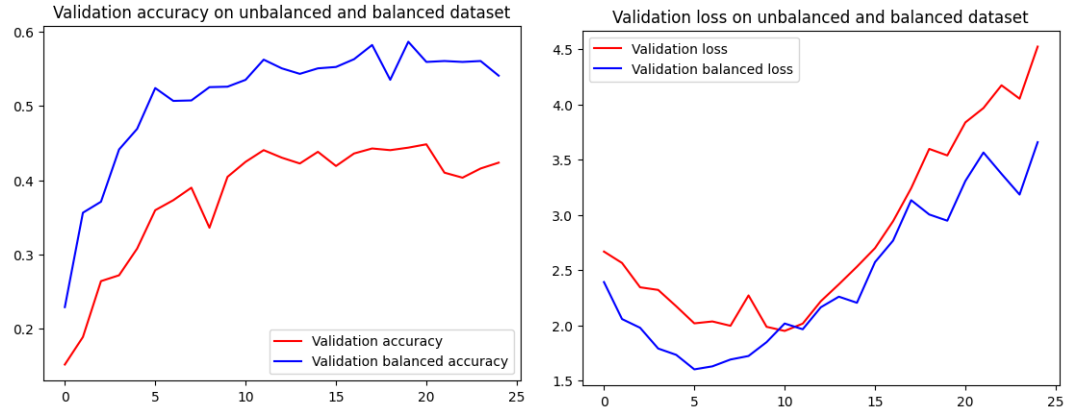


Figure 14: Plot of the validation accuracy/loss of the unbalanced dataset vs balanced dataset

These is what we were expecting, since we've used data augmentation to enrich our starting dataset with new samples with the aim of reducing the overfitting.
The last step was to compare the test set results, also in this case we can see an improvement with the usage of the balanced dataset:

- Using the unbalanced the accuracy was: **0.4245**
- Using the balanced the accuracy was: **0.4505**

After the comparison of the two dataset, we've decided to continue with the balanced dataset due to the improvements encountered.
Starting from this model we will build different model trying to increase the accuracy of it introducing new layers with different configuration of hyperparameters and changing the architecture of the network.

4.7 Second model: adding the dropout layer

The second model mantains the general architecture seen before, so an initial rescaling layer, 5 conv2D layers alterned by 4 MaxPooling2D layers and finally a flatten layer. Our principal concern was about overfitting, in fact, we have tried to reduce it with further improvements of the starting model. In this second model we have added the dropout; it is a regularisation technique applied to the output features of a layer, in our case the flatten layer; it consists in *dropping out* some features, by setting them to zero, following a random behaviour that depends on the *dropout rate*, which is the fraction of features that are set to zero. Normally, the rate belongs to the interval [0.2, 0,5]; we have selected for our model an initial value for this parameter equal to **0.3**. Basically we've reduced the units active during the training by a factor of 0.3. This help us in reducing the overfitting since the values equal to zero in the output layer avoid the memorising of their precedent values without learning something, which is the reason of the loss of the capacity to generalise of a model.
The dropout can be added using the dropout layer:

```
x = layers.Dropout(0.3)(x)
```

In the following figure 15 we can see the model summary of the second model:

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[(None, 180, 180, 3)]	0
rescaling_1 (Rescaling)	(None, 180, 180, 3)	0
conv2d_5 (Conv2D)	(None, 178, 178, 32)	896
max_pooling2d_4 (MaxPooling 2D)	(None, 89, 89, 32)	0
conv2d_6 (Conv2D)	(None, 87, 87, 64)	18496
max_pooling2d_5 (MaxPooling 2D)	(None, 43, 43, 64)	0
conv2d_7 (Conv2D)	(None, 41, 41, 128)	73856
max_pooling2d_6 (MaxPooling 2D)	(None, 20, 20, 128)	0
conv2d_8 (Conv2D)	(None, 18, 18, 256)	295168
max_pooling2d_7 (MaxPooling 2D)	(None, 9, 9, 256)	0
conv2d_9 (Conv2D)	(None, 7, 7, 256)	590080
flatten_1 (Flatten)	(None, 12544)	0
dropout_1 (Dropout)	(None, 12544)	0
dense_1 (Dense)	(None, 18)	225810

Total params: 1,204,306
Trainable params: 1,204,306
Non-trainable params: 0

Figure 15: Added a dropout layer to the model

We've trained the model for *50* epochs, the plot of the accuracies of training and validation and the plot of their losses are depicted in figure 16.

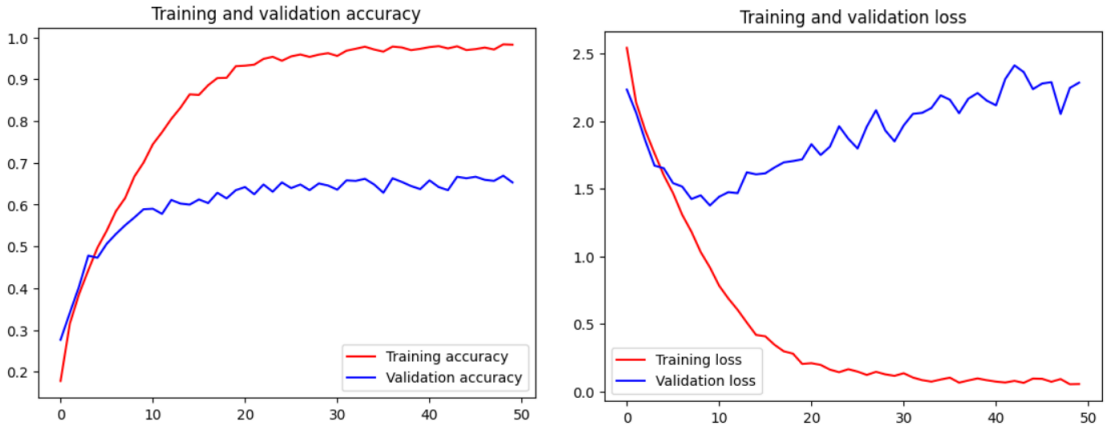


Figure 16: Training and validation accuracy and loss plots

As we can see we've an improvement with respect to the previous model, while the latter has reached a validation accuracy that stalls at 55%, the first obtains better results reaching a validation accuracy of 65% with an increment of 10%.
The training accuracy stalls at 98% in the model with dropout, while in the previous model we've observed an accuracy of 97%.
The same can be seen through the loss plots, the increasing in the validation loss of the second model is less abrupt respect the growth of the validation loss of the previous model, consequently the first reaches a loss equal to **3.66** at end of the training and the second reaches **2.28** with a significant reduction.
A summary of the values of loss and test during the different parts of the building of model is presented in the table 3.

CNN with Dropout		
Number of Epochs:	50	
Dropout rate:	0.3	
	Loss	**Accuracy**
Train	0.0559	0.9827
Validation	2.2858	0.6531
Test	4.3013	0.5081

Table 3: Results achieved with the second model

As last point we can observe an improvement also in the test accuracy that is increased from **0.4505** to **0.5081**.

The following step that we've done was to change the *dropout rate* with values inside the interval [0.2, 0.5], as consequence of these tests we've found that the best improvement is obtained with a rate of **0.5**.

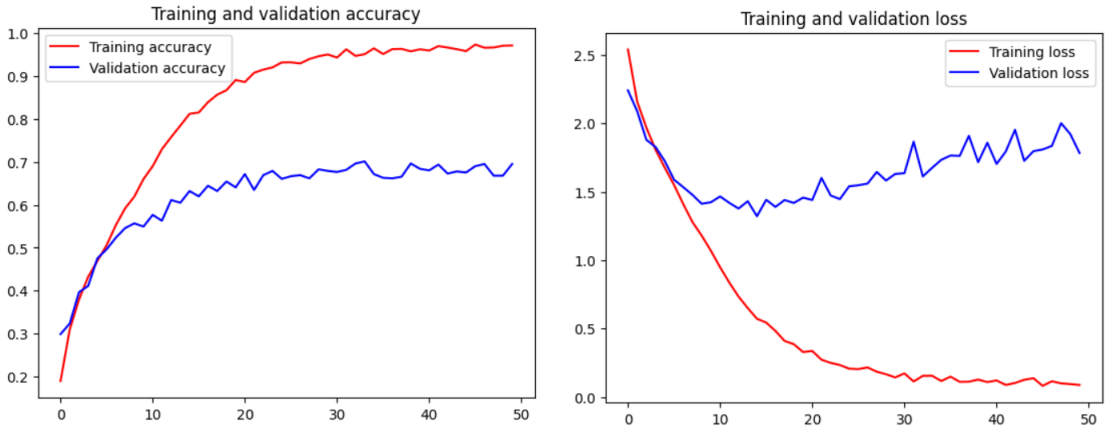


Figure 17: Training and validation accuracy and loss plots with a dropout rate of 0.5

CNN with Dropout		
Number of Epochs:	50	
Dropout rate:	0.5	
	Loss	**Accuracy**
Train	0.0875	0.9713
Validation	1.7832	0.6951
Test	3.4581	0.5171

Table 4: Results achieved with the second model increasing the dropout rate to 0.5

We can see from figure 17 that the plot is similar to the one seen in figure 16, so the model is affected by overfitting, but we've an increase in the accuracy both on test and validation: the accuracy during the validation using 0.5 as dropout rate is 69% that is better than the previous value of 65%.
Another improvement is in the losses of both the validation and the test results: the loss on validation is reduced from **2.2858** to **1.7832** while the loss on the test is decreased from **4.3013** to **3.4581**

In conclusion the introduction of the dropout in our model helps us to handle the overfitting, in fact we've reduced it using this model, but it was present since the model lose the capacity to generalize around the epoch 10 and it corresponds to the stall of the validation accuracy's plot while the training accuracy converges to 100%.

4.8 Trying different architectures with dropout layer

In this chapter we report the results obtained with two different network architectures. We added and removed layers of the network and changed the number of hidden units for each layer.

4.8.1 First experiment with fewer training parameters

In this experiment we reduced the number of parameters of the network. We modified the previous model adding a Conv2D layer right after the third Conv2D layer, but the main difference from the previous model that reduced drastically the final number of parameters is that we removed the last Conv2D layer which had 590080 parameters. In this way the final number of trainable parameters was reduced to 338,290 from the previous 1,204,306 parameters.

In the following figure 18 we can see the model summary:

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 180, 180, 3)]	0
rescaling (Rescaling)	(None, 180, 180, 3)	0
conv2d (Conv2D)	(None, 178, 178, 32)	896
max_pooling2d (MaxPooling2D)	(None, 89, 89, 32)	0
conv2d_1 (Conv2D)	(None, 87, 87, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 43, 43, 64)	0
conv2d_2 (Conv2D)	(None, 41, 41, 128)	73856
conv2d_3 (Conv2D)	(None, 39, 39, 128)	147584
max_pooling2d_2 (MaxPooling2D)	(None, 19, 19, 128)	0
conv2d_4 (Conv2D)	(None, 17, 17, 64)	73792
max_pooling2d_3 (MaxPooling2D)	(None, 8, 8, 64)	0
conv2d_5 (Conv2D)	(None, 6, 6, 32)	18464
max_pooling2d_4 (MaxPooling2D)	(None, 3, 3, 32)	0
flatten (Flatten)	(None, 288)	0
dropout (Dropout)	(None, 288)	0
dense (Dense)	(None, 18)	5202

Total params: 338,290
Trainable params: 338,290
Non-trainable params: 0

Figure 18: Changed architecture to the model

We've trained the model for *50* epochs, the plot of the accuracies of training and validation and the plot of their losses are depicted in figure 19.

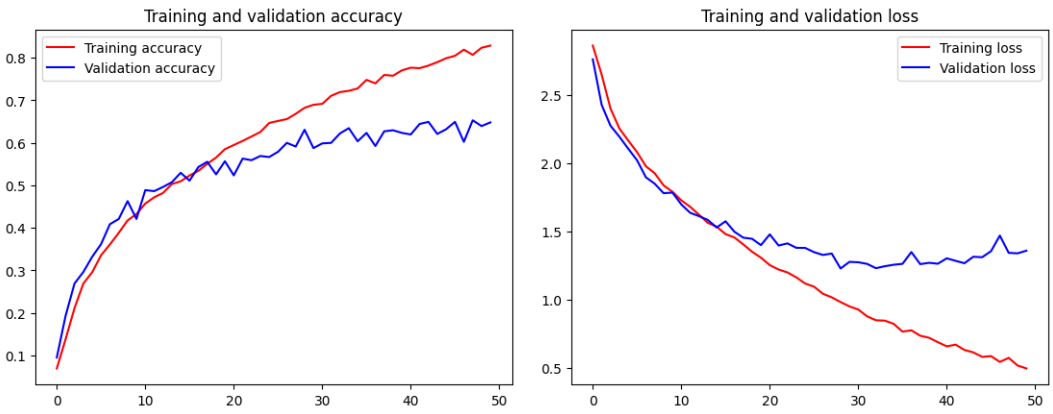


Figure 19: Training and validation accuracy and loss plots

Looking at the plots we can notice how the blue and red curves are a lot closer compared to the second model (figure 16), for about 20 epochs the validation and training accuracy curves are approximately overlapping while in the second model they overlapped for only 8 epochs. Also the loss curves behave similarly. This means that there is an improvement in fighting overfitting.

CNN with fewer parameters		
Number of Epochs:	50	
Dropout rate:	0.5	
	Loss	**Accuracy**
Train	0.4558	0.8492
Validation	1.4210	0.6432
Test	2.2602	0.5171

Table 5: Results achieved with the model

The results reported in this table are not really better from the previous model. The validation accuracy decreased from **69%** to **64%** while the validation loss is better since it decreased from **1.78** to **1.42**. On the test set this model performed a bit better since the test accuracy is the same (**51.71%**) but the test loss decreased from **3.4581** to **2.2602**.

4.8.2 Second experiment adding a Dropout and Dense layer

In this experiment we considered the previous model and added a Dense and Dropout layers after the first Dropout layer. We report the code used for the last layers to better understand:

```
x = layers.Dropout(0.5)(x)
x = layers.Dense(256, activation="relu")(x)
x = layers.Dropout(0.25)(x)
outputs = layers.Dense(18, activation="softmax")(x)
```

We can see from the code that we tried a different dropout rate (of 0.25) for the Dropout layer and a different activation function for the Dense layer (*relu* activation). We wanted to see how these changes affected the results, the first observation is that the number of trainable parameters increased to 1,720,756, which is even greater than the number of parameters of the second model (1,204,306).

In the following figure 20 we can see the model summary:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 180, 180, 3)]     0

rescaling_1 (Rescaling)      (None, 180, 180, 3)       0

conv2d_6 (Conv2D)            (None, 178, 178, 32)      896

max_pooling2d_5 (MaxPooling  (None, 89, 89, 32)        0
2D)

conv2d_7 (Conv2D)            (None, 87, 87, 64)        18496

max_pooling2d_6 (MaxPooling  (None, 43, 43, 64)        0
2D)

conv2d_8 (Conv2D)            (None, 41, 41, 128)       73856

conv2d_9 (Conv2D)            (None, 39, 39, 128)       147584

max_pooling2d_7 (MaxPooling  (None, 19, 19, 128)       0
2D)

conv2d_10 (Conv2D)           (None, 17, 17, 256)       295168

max_pooling2d_8 (MaxPooling  (None, 8, 8, 256)         0
2D)

conv2d_11 (Conv2D)           (None, 6, 6, 256)         590080

max_pooling2d_9 (MaxPooling  (None, 3, 3, 256)         0
2D)

flatten_1 (Flatten)          (None, 2304)              0

dropout_2 (Dropout)          (None, 2304)              0

dense_2 (Dense)              (None, 256)               590080

dropout_3 (Dropout)          (None, 256)               0

dense_3 (Dense)              (None, 18)                4626

=================================================================
Total params: 1,720,786
Trainable params: 1,720,786
Non-trainable params: 0
_________________________________________________________________
```

Figure 20: Changed architecture to the model

We've trained the model for *75* epochs, the plot of the accuracies of training and validation and the plot of their losses are depicted in figure 21.

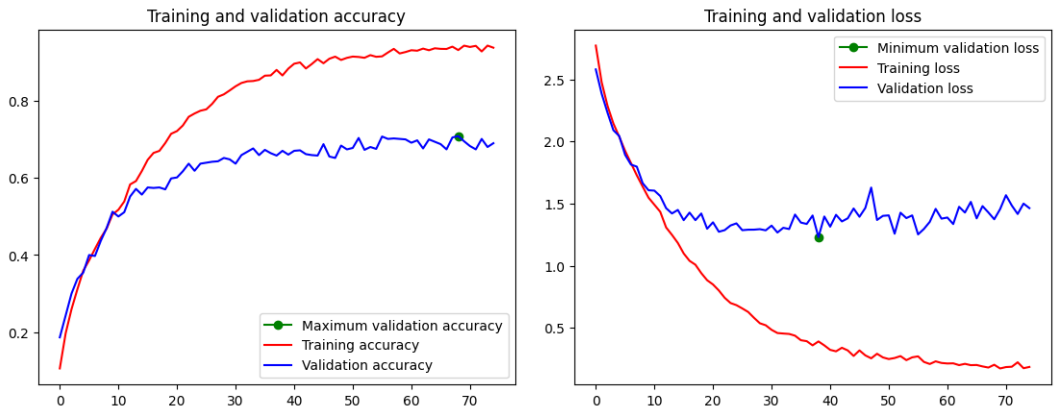


Figure 21: Training and validation accuracy and loss plots

Comparing these plots with the ones in figure 17 we can see that even in this architecture there is an improvement in fighting overfitting since the blue and red curves are closer than before. The training accuracy reaches nearly 100% in about 70 epochs, while in the second model in 50 epochs. Looking at the loss plot we can also notice how the validation curve stalls near **1.5** after 20 epochs, without increasing in the following epochs as it happened in the second model.

Larger CNN		
Number of Epochs: 75		
	Loss	**Accuracy**
Train	0.1860	0.9379
Validation	1.4642	0.6901
Test	2.8212	0.5090

Table 6: Results achieved with the model

Comparing this table with table 4 we notice how the validation and test accuracy are approximately the same (**0.69** and **0.51**), but the losses are lower than before and that's better! In fact validation loss is **1.46** and the test loss is **2.82**, in table 4 they were **1.78** and **3.46**. This means that this model generalizes a bit better than the second model!

Comparing this architecture with the first experiment we don't see many differences, the validation accuracy is better (**69%** while in the first experiment was **64%**), the test loss is a bit worse (**2.82** while in the first experiment it was **2.26**), the other values are approximately the same. We can still prefer this model to the first experiment for the higher validation accuracy, which is the aspect we are now aiming to improve the most.

4.9 Adding L1 L2 weight regularization

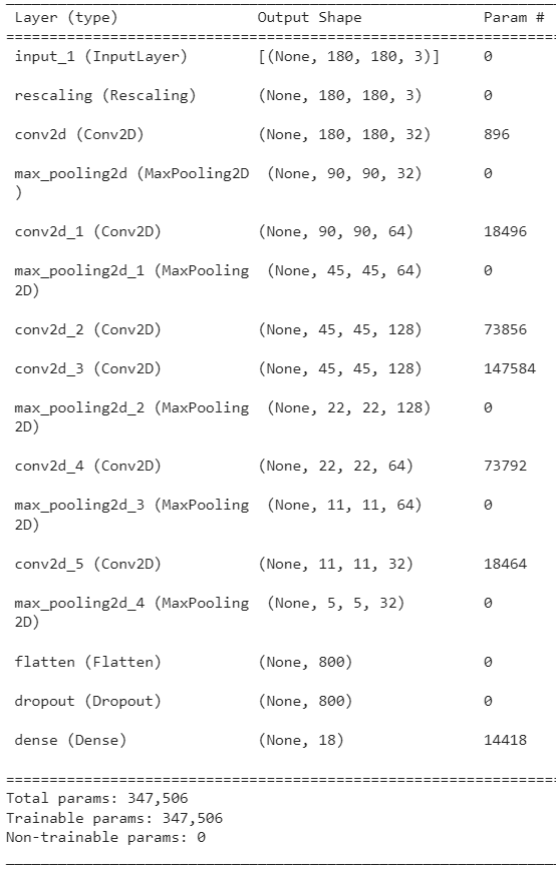
In this chapter we report the results of the experiments obtained after adding weight regularization to some of the previous models.

4.9.1 Keras callbacks

We also decided to add the following Keras callbacks:

- **Early Stopping:** this callback lets us interrupt training when validation loss has stopped improving for more than 10 epochs, so as soon as the model starts overfitting.
- **Model Checkpoint:** this callback lets us continously save the model after every epoch during training. We chose the option to only save the best model so far, the one with the best val loss
- **ReduceLROnPlateau:** this callback lets us reduce the learning rate when the validation loss has stopped improving, which is an effective strategy to get out of local minima during training.

For the first two experiments we decided to use the model with fewer parameters but setting padding=*same*, because in general padding allows CNN to make a more accurate analysis of images. The number of parameters increased to 347,505, which isn't much greater than the model with padding=*valid*, so we don't expect this parameter to make much of a difference in the results. In the following figure 22 we can see the model summary:



Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 180, 180, 3)]	0
rescaling (Rescaling)	(None, 180, 180, 3)	0
conv2d (Conv2D)	(None, 180, 180, 32)	896
max_pooling2d (MaxPooling2D)	(None, 90, 90, 32)	0
conv2d_1 (Conv2D)	(None, 90, 90, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 45, 45, 64)	0
conv2d_2 (Conv2D)	(None, 45, 45, 128)	73856
conv2d_3 (Conv2D)	(None, 45, 45, 128)	147584
max_pooling2d_2 (MaxPooling2D)	(None, 22, 22, 128)	0
conv2d_4 (Conv2D)	(None, 22, 22, 64)	73792
max_pooling2d_3 (MaxPooling2D)	(None, 11, 11, 64)	0
conv2d_5 (Conv2D)	(None, 11, 11, 32)	18464
max_pooling2d_4 (MaxPooling2D)	(None, 5, 5, 32)	0
flatten (Flatten)	(None, 800)	0
dropout (Dropout)	(None, 800)	0
dense (Dense)	(None, 18)	14418

Total params: 347,506
Trainable params: 347,506
Non-trainable params: 0

Figure 22: Choosen model summary

4.9.2 Experiment with L1 and L2 regularizations in the Dense layer

As we said before for this experiment we added some Keras callbacks, the code used is:

```
callbacks_list = [
keras.callbacks.EarlyStopping(
monitor='val_loss',
patience=10,
),
keras.callbacks.ModelCheckpoint(
filepath=os.path.join(dir_name, 'L2_model.h5'),
monitor='val_loss',
save_best_only=True,
),
keras.callbacks.ReduceLROnPlateau(
monitor='val_loss',
factor=0.1,
patience=9,
)
]
```

For all the callbacks we decided to monitor the metric validation loss instead of the validation accuracy. In the previous experiments both the validation and the loss curves

didn't have many oscillations, so we believe that there will be not much difference in the choice of the metric to monitor. For *EarlyStopping* we set the parameter *patience* to 10, in this way when the validation loss is not improving after 10 epochs the model stops the training. These are just some initial values and maybe the model won't be stopped since 10 maybe is too big, we will modify it if the results won't be satisfying. The patience of *ReduceLROnPlateau* is initially set to 9, in this way the learning rate of the optimizer will be changed by a factor of 0.1 before EarlyStopping stops the training at the 10 epoch.

For this first experiment we used L1 and L2 regularization only in the dense layer of our model. We imported the functions l1,l2 and l1_l2 from *keras.regularizers*:

```
from keras.regularizers import l1, l2, l1_l2
```

and added the *kernel_regularizer* parameter to the Dense layer in this way:

```
outputs = layers.Dense(18, activation="softmax",
    kernel_regularizer=l1_l2(l1=0.02, l2=0.05))(x)
```

We've trained the model for *70* epochs and noticed that Early Stopping didn't interrupt it. Also the learning rate of Adam optimizer didn't change from the initial value 0.01.

The plot of the accuracies of training and validation and the plot of their losses are depicted in figure 24.

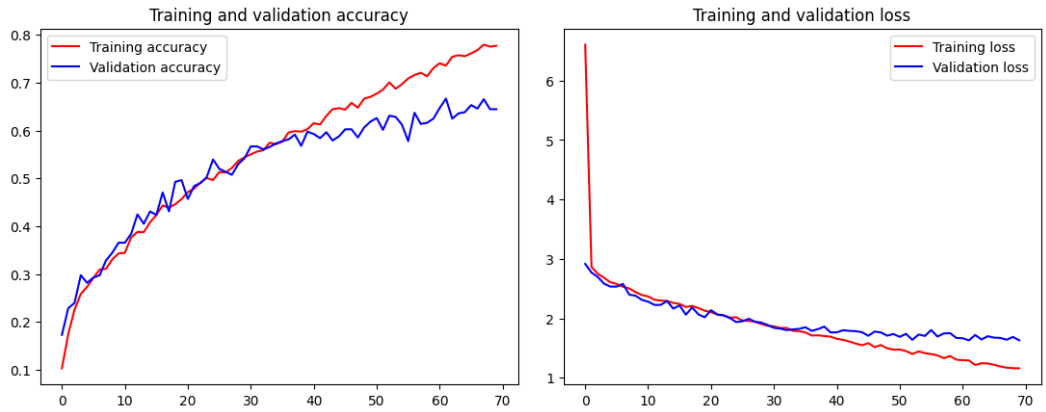


Figure 23: Training and validation accuracy and loss plots

From the plots we can immediately notice that the training and validation accuracy curves are a lot closer than in the previous models, it's a good sign against overfit! The maximum validation accuracy and the minimum validation loss reached are approximately the same as before.

4.9.3 Experiment with the same model as before and L2 regularization in the first Conv2D layer

In this experiment we considered the model of the first experiment, with L1 and L2 regularization in the dense layer, and added L2 regularization in the first Conv2Dlayer in this way:

```
x = layers.Conv2D(filters=32, kernel_size=3, strides=1, activation="relu",
    padding="same", kernel_regularizer=l2(0.01))(x)
```

We also decided to decrease the patience of the callback *ReduceLROnPlateau* to 5 since in the previous experiment the learning rate of Adam optimizer didn't change.

The plot of the accuracies of training and validation and the plot of their losses are depicted in figure 24.

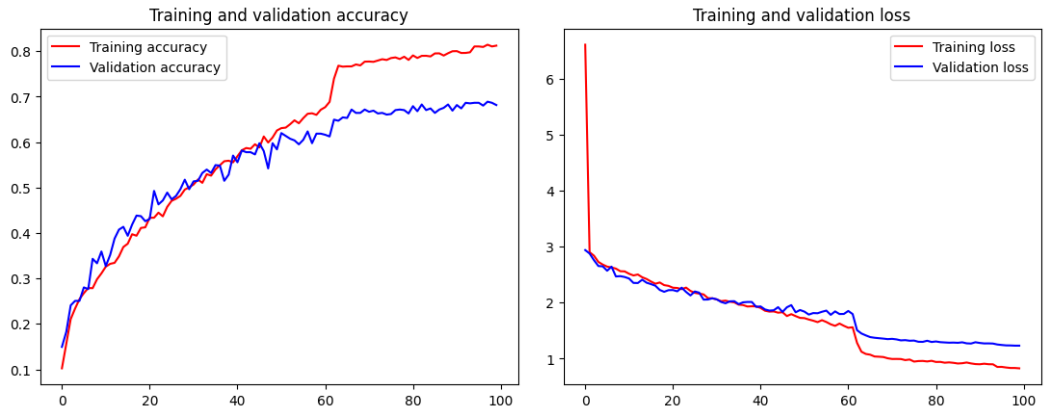


Figure 24: Training and validation accuracy and loss plots

Even in this experiment the blue and red curves are very close! The model starts overfitting only after 60 epochs. At epoch 65 the learning rate of the optimizer changed from 10^{-3} to 10^{-4}, and this is why in both of the plots we can notice an improvement in the accuracy and in the loss curves. Also in this second experiment Early Stopping didn't interrupt the training and all the 100 epochs were used to train.

CNN with L1 L2 regularization		
Number of Epochs:	100	
	Loss	**Accuracy**
Train	0.8346	0.8143
Validation	1.2369	0.6889
Test	1.6898	0.5899

Table 7: Results achieved with the fourth model

Comparing this table with the results of the model without weight regularization in table 5 we can see clear improvements in the generalization power of the model! The validation loss decreased from **1.42** to **1.24**, and the validation accuracy increased from **64%** to nearly **69%**. Also, the test loss decreased from **2.26** to **1.69** and the test accuracy increased from **52%** to **59%**.

4.9.4 Experiments changing model

Since the results in the previous experiments were very satisfying, we decided to use weight regularization in a different model and see how it behaves. We maintained the same values of l1 and l2 both in the dense layer and in the first Conv2D layer, in the same way as before.

4.9.5 First experiment with a larger number of training parameters, padding=*same*

For this experiment we used the model in section 4.8.2 with the only modification of the padding value to *same*. As we said before in general this value helps the CNN to be more accurate, but in this case the number of parameters increased significantly with respect to the model shown in figure 20, from **1,720,786** to **2,769,362**. This experiment is also interesting to see how the great number of parameters will affect the performances.

In the following figure 25 we can see the model summary of the model:

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 180, 180, 3)]	0
rescaling (Rescaling)	(None, 180, 180, 3)	0
conv2d (Conv2D)	(None, 180, 180, 32)	896
max_pooling2d (MaxPooling2D)	(None, 90, 90, 32)	0
conv2d_1 (Conv2D)	(None, 90, 90, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 45, 45, 64)	0
conv2d_2 (Conv2D)	(None, 45, 45, 128)	73856
conv2d_3 (Conv2D)	(None, 45, 45, 128)	147584
max_pooling2d_2 (MaxPooling2D)	(None, 22, 22, 128)	0
conv2d_4 (Conv2D)	(None, 22, 22, 256)	295168
max_pooling2d_3 (MaxPooling2D)	(None, 11, 11, 256)	0
conv2d_5 (Conv2D)	(None, 11, 11, 256)	590080
max_pooling2d_4 (MaxPooling2D)	(None, 5, 5, 256)	0
flatten (Flatten)	(None, 6400)	0
dropout (Dropout)	(None, 6400)	0
dense (Dense)	(None, 256)	1638656
dropout_1 (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 18)	4626

Total params: 2,769,362
Trainable params: 2,769,362
Non-trainable params: 0

Figure 25: Changed architecture to the model

In this experiment we set the patience of *EarlyStopping* to 8, a bit smaller than before. The patience of *ReduceLROnPlateau* is still 5. We've trained the model for *100* epochs, the plot of the accuracies of training and validation and the plot of their losses are depicted in figure 26.

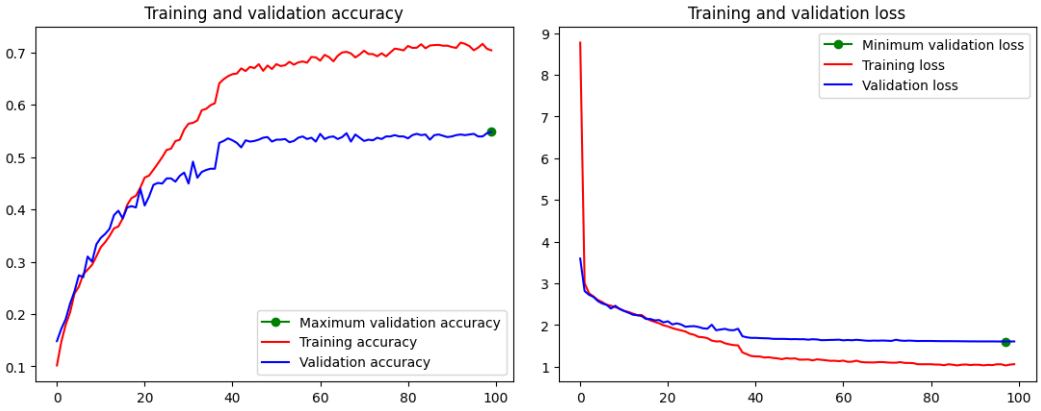


Figure 26: Training and validation accuracy and loss plots

Even in this case from the plots we can notice that the training and validation accuracy curves are closer than in the model of Chapter 4.8.2. However there is a significant deterioration in the accuracy performance, in the previous model the accuracy reached **69%**, but in this experiment it stalls at **55%**. This means that the generalization power of this model worsened, and weight regularization is not enough to limit the overfitting caused by the higher number of parameters of this model. This is why we did a last experiment with this same model but setting padding to *valid*, so diminishing the number of parameters.

4.9.6 Second experiment with padding=*valid*

In the following figure 27 we can see the model summary of the model:

Layer (type)	Output Shape	Param #
input_1 (InputLayer)	[(None, 180, 180, 3)]	0
rescaling (Rescaling)	(None, 180, 180, 3)	0
conv2d (Conv2D)	(None, 178, 178, 32)	896
max_pooling2d (MaxPooling2D)	(None, 89, 89, 32)	0
conv2d_1 (Conv2D)	(None, 87, 87, 64)	18496
max_pooling2d_1 (MaxPooling2D)	(None, 43, 43, 64)	0
conv2d_2 (Conv2D)	(None, 41, 41, 128)	73856
conv2d_3 (Conv2D)	(None, 39, 39, 128)	147584
max_pooling2d_2 (MaxPooling2D)	(None, 19, 19, 128)	0
conv2d_4 (Conv2D)	(None, 17, 17, 256)	295168
max_pooling2d_3 (MaxPooling2D)	(None, 8, 8, 256)	0
conv2d_5 (Conv2D)	(None, 6, 6, 256)	590080
max_pooling2d_4 (MaxPooling2D)	(None, 3, 3, 256)	0
flatten (Flatten)	(None, 2304)	0
dropout (Dropout)	(None, 2304)	0
dense (Dense)	(None, 256)	590080
dropout_1 (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 18)	4626

Total params: 1,720,786
Trainable params: 1,720,786
Non-trainable params: 0

Figure 27: Changed architecture to the model

The callbacks are setted in the same way as before, we report the code to recap:

```
callbacks_list = [
keras.callbacks.EarlyStopping(
monitor='val_loss',
patience=8,
),
keras.callbacks.ModelCheckpoint(
filepath=os.path.join(dir_name, 'model2_regularization.h5'),
monitor='val_loss',
save_best_only=True,
),
keras.callbacks.ReduceLROnPlateau(
monitor='val_loss',
factor=0.1,
patience=5,
)
]
```

We've trained the model for *100* epochs, the plot of the accuracies of training and validation and the plot of their losses are depicted in figure 28.

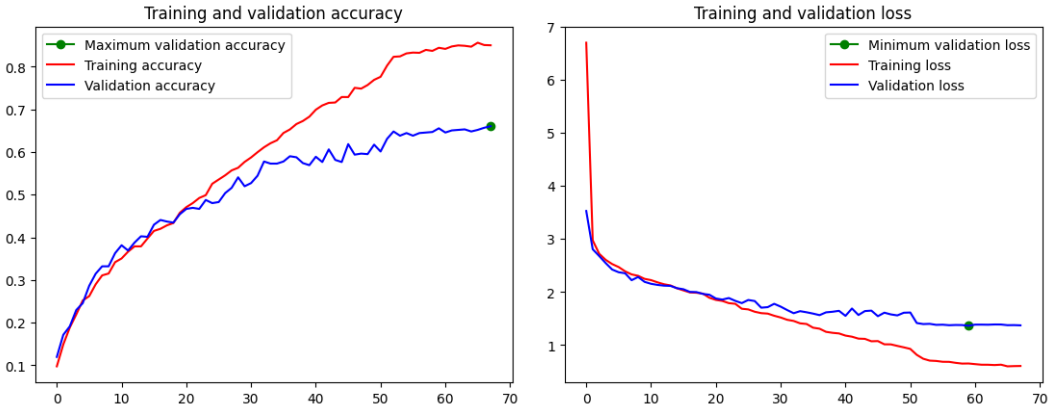


Figure 28: Training and validation accuracy and loss plots

Even in this experiment the blue and red curves are closer than the model in chapter 4.8.2. At epoch 52 the learning rate changed from 10^{-3} to 10^{-4}, this is why in the plots both the accuracy and the loss suddenly improved. In this experiment the callback *EarlyStopping* stopped the training of the model at epoch 68 and not at epoch 100, in fact at epoch 60 the validation loss reached **1.3653**, which is the minimum in the remaining 8 epochs. The model was saved at epoch 60 as specified by the callback *Model Checkpoint*.

second CNN with L1 L2 regularization		
Number of Epochs:	68	
	Loss	**Accuracy**
Train	0.6547	0.8443
Validation	1.3653	0.6556
Test	2.0179	0.5297

Table 8: Results achieved with the model

In this experiment the validation accuracy performed a lot better than in the first experiment with more than 2,769,362. This confirms the hypothesis that too many trainable parameters didn't let the model generalize much. Comparing the results with the ones reported in the table in chapter 4.8.2 we observe that the validation accuracy (**66%**)is worse than before (**69%**), the validation loss improved but not significantly (from **1.46** to **1.37**). However on the test set the results of this experiment are better than before! The test accuracy now reaches nearly **53%** when it was **51%**, the test loss decreased from **2.82** to **2.02**.

We can conclude that weight regularization made a great difference in the first model to fight overfit, while in the second model the improvment is not so clear since the validation accuracy decreased a bit.

5 Visualization

In this chapter we'll show the application of two visualization techniques to one of the saved CNN from scratch models. These techniques help to visualize and interpret how the model represents the concepts learned on the test images, in this way the model is no more a "black box" and it's easier for humans to debug the model.

The visualization techniques applied are:

- **Visualizing intermediate convnet outputs (intermediate activations)**: useful for understanding how successive convent layers transform their input, and how it is decomposed into the different filters learned by the network.
- **Visualizing heatmaps of class activation in an image**: useful for understanding which parts of an image were identified as belonging to a given class.

We decided to use the model depicted in figure 22, it was one of the best performing CNN from scratch, it reached an accuracy of 59% on the test set and was composed of 5 convolutional layers.

Then we selected an image present in the test set to feed into our model, the image belongs to the class *Vincent van Gogh* and was classified correctly by the model.

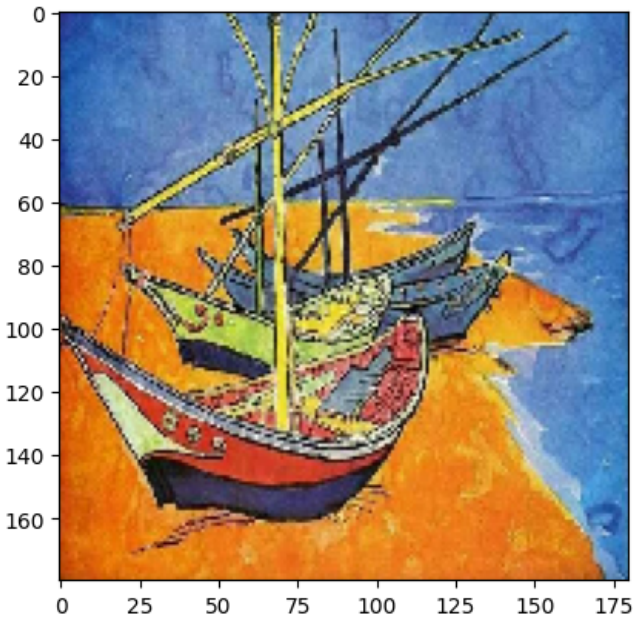


Figure 29: Test image

5.1 Visualizing intermediate activations

Visualizing intermediate activations consists of displaying the feature maps that are output by various convolution and pooling layers in a network, given a certain input. To extract the feature maps we used a Keras model that takes the test image as input and outputs the activations of all convolution and pooling layers.

The following figure shows 3 channels of the activation of the first convolution layer (*conv2d*) for the boat image input:

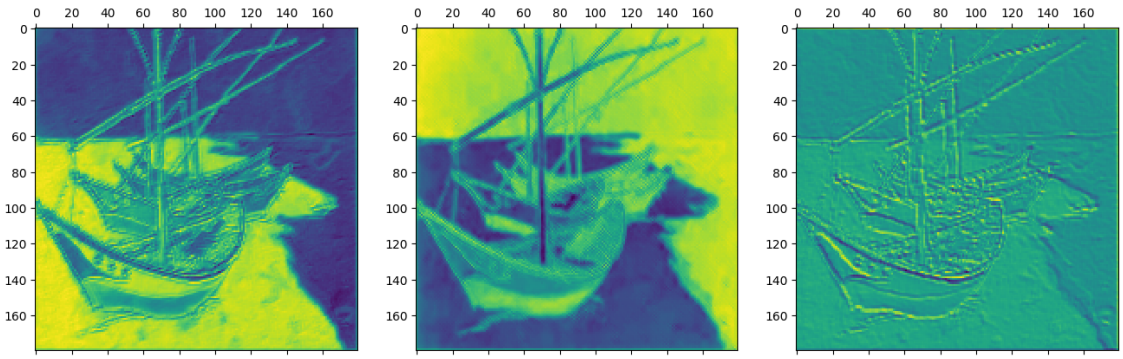
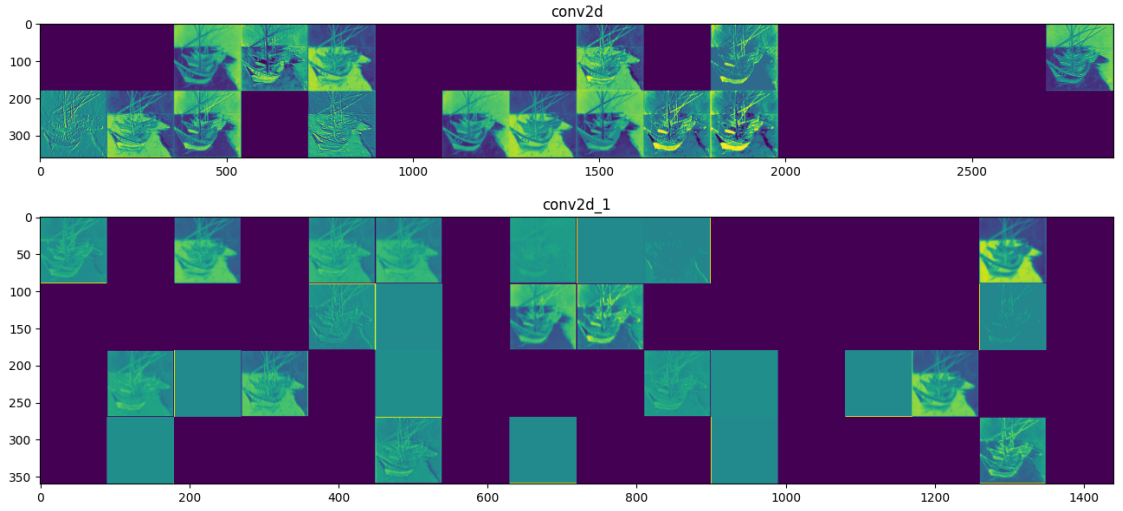


Figure 30: 3 channels of the activation of the first convolutional layer on the test picture

We tried to give an interpretation of these channels, for example in the first two channels the model probably captured a color filter, the first channel a red detector, and the second one a blue detector. The last picture is more difficult to interpret but it seems like this filter scans the contours of the figures present in the image (edge detector).

At this point we plotted a complete visualization of all the activations in the network. In figure 31 we can see every channel in each of the activation maps of the first three convolutional layers stacked side by side. The blank filters mean that in the image the pattern encoded in the image wasn't found.



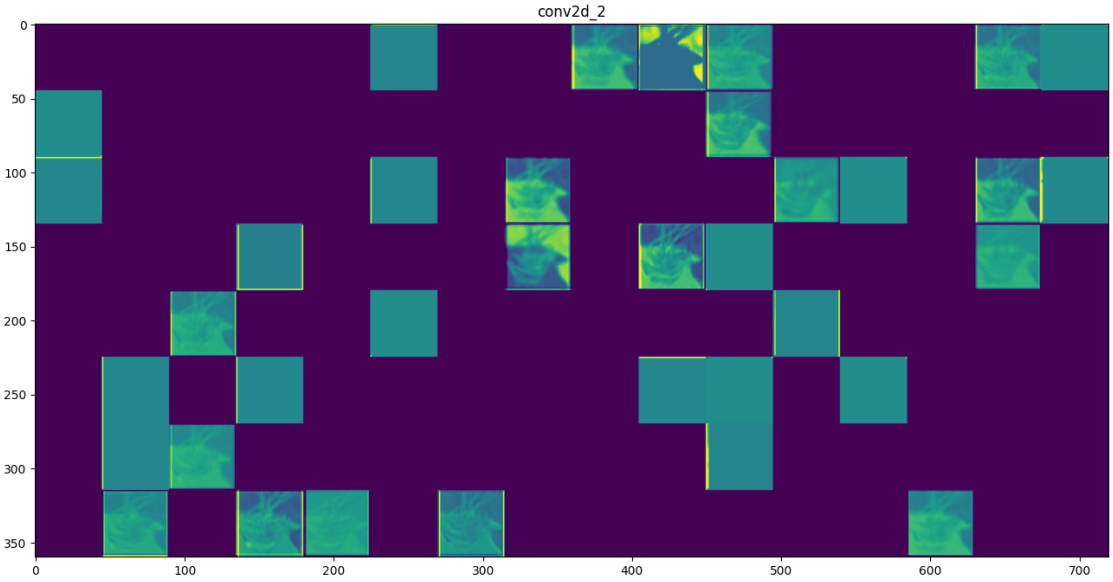


Figure 31: activations of intermediate conv layers

We notice how the activations of the first layer are the most easily interpretable, in the higher layers they become too abstract since they carry increasingly less information about the visual contents of the image and increasingly more information related to the class of the image.

5.2 Visualizing heatmaps of class activation

Visualizing heatmaps of class activation is useful to understand which parts of a given image led a convnet to its final classification decision. For this part we selected another image of *Vincent van Gogh* that the model still classified correctly.

The second input image is:

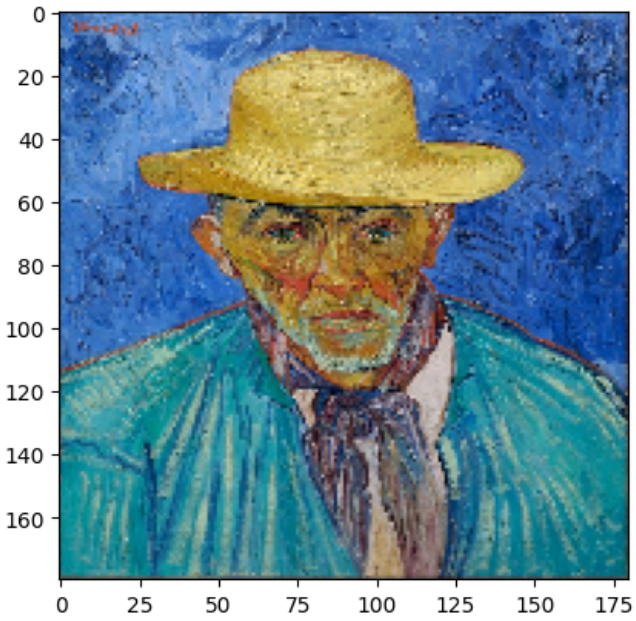


Figure 32: Second test image

In the activation heatmap we can see how much a part of the input images is important for the considered class, in our case it shows what the model looks at when it classifies our images to the class *Vincent van Gogh*.

The heatmaps of the two input images are depicted in the following figure:

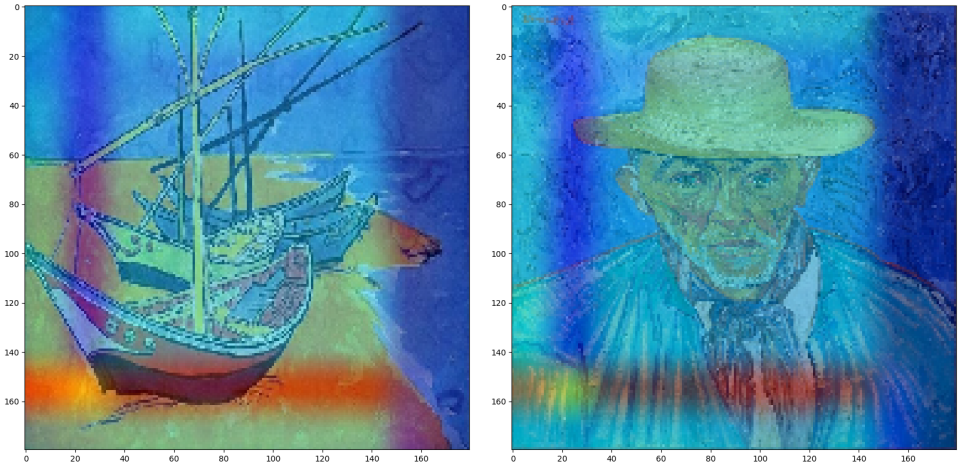


Figure 33: Superimposing the class activation heatmaps on the original pictures

We notice how this model analyzes only a strip of the images (the red-colored part) instead of focusing on their content. This probably means that it looks at the texture of the paintings, the way the artist used their brushes, and so on. Thanks to that the model recognized correctly the artist, however we don't know if this model is robust in case of a change of the texture in the same part of the image, for example if the painting is a bit ruined by the agents of the time.

6 Pre-trained CNN

In this section we'll report the results and observations from using different pre-trained CNNs to solve the task of artist recognition. To better understand how we have worked let's remember that convnets consist of two parts:

- **a convolutional base** (series of convolutional and pooling layers)
- **a densely connected classifier**

In general there are two ways to leverage a pre-trained network:

- **feature extraction**: the idea is to take the features extracted by the convolutional base of the pre-trained CNN and run them through a new classifier trained from scratch.

- **fine-tuning**: it consists of unfreezing a few of the top layers of a frozen model base used for feature extraction, and jointly training both the newly added fully connected classifier and these top layers.

In all of the experiments reported we followed the same steps for feature extraction and fine-tuning, in particular the first 3 steps are the same for both of them. The steps for fine-tuning are:

1. We froze the already trained base network of the selected pre-trained CNN.
2. We added a custom classifier on top of it.
3. We trained the part we added.
4. We unfreezed some layers in the base network.
5. We jointly trained both these layers and the part added.

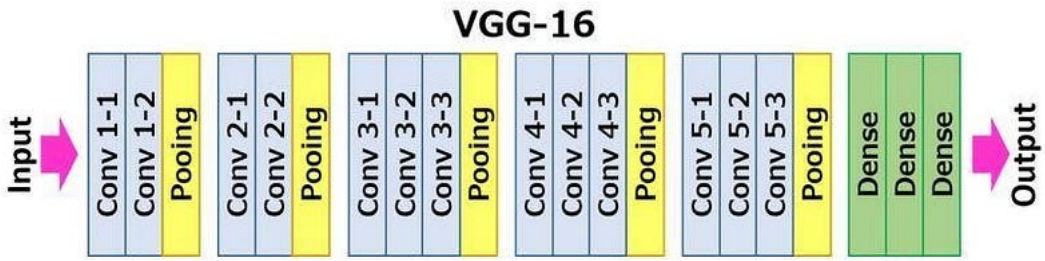
For the experiments we decided to use the following pre-trained CNNs:

- VGG16
- ResNet50V2
- Xception
- InceptionV3

6.1 VGG16

In this chapter we'll report the results obtained using VGG16, a network pre-trained on ImageNet. In VGG16 there are 13 convolutional layers, 5 Max Pooling layers, and 3 Dense layers which sum up to 21 layers but it has only sixteen weight layers i.e., the learnable parameters layer.

In the following picture we can see VGG16's architecture:



6.1.1 VGG16 Feature Extraction

Firstly we instantiated the VGG16 convolutional base, the code used is:

```
from tensorflow.keras.applications import VGG16

conv_base = keras.applications.vgg16.VGG16(
    weights="imagenet",
    include_top=False,
    input_shape=(180, 180, 3))
```

Let's explain the values of the parameters used:

- **weights** = "imagenet". This argument specifies the weight checkpoint from which to initialize the model.
- **include_top** = False. This argument refers to including (or not) the densely connected classifier on top of the network. We chose not to include it since we will use our own classifier trained from scratch.
- **input_shape** = (180, 180, 3). This is the shape of the input tensors of the network. In fact as before we use images with *height*=*width*=180 and 3 channels since they are RGB images.

This VGG16 convolutional base summary is depicted in figure 34.

```
conv_base.summary()
```

```
Model: "vgg16"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_1 (InputLayer)        [(None, 180, 180, 3)]     0

 block1_conv1 (Conv2D)       (None, 180, 180, 64)      1792

 block1_conv2 (Conv2D)       (None, 180, 180, 64)      36928

 block1_pool (MaxPooling2D)  (None, 90, 90, 64)        0

 block2_conv1 (Conv2D)       (None, 90, 90, 128)       73856

 block2_conv2 (Conv2D)       (None, 90, 90, 128)       147584

 block2_pool (MaxPooling2D)  (None, 45, 45, 128)       0

 block3_conv1 (Conv2D)       (None, 45, 45, 256)       295168

 block3_conv2 (Conv2D)       (None, 45, 45, 256)       590080

 block3_conv3 (Conv2D)       (None, 45, 45, 256)       590080

 block3_pool (MaxPooling2D)  (None, 22, 22, 256)       0

 block4_conv1 (Conv2D)       (None, 22, 22, 512)       1180160

 block4_conv2 (Conv2D)       (None, 22, 22, 512)       2359808

 block4_conv3 (Conv2D)       (None, 22, 22, 512)       2359808

 block4_pool (MaxPooling2D)  (None, 11, 11, 512)       0

 block5_conv1 (Conv2D)       (None, 11, 11, 512)       2359808

 block5_conv2 (Conv2D)       (None, 11, 11, 512)       2359808

 block5_conv3 (Conv2D)       (None, 11, 11, 512)       2359808

 block5_pool (MaxPooling2D)  (None, 5, 5, 512)         0

=================================================================
Total params: 14,714,688
Trainable params: 14,714,688
Non-trainable params: 0
```

Figure 34: Training and validation accuracy and loss plots

We can see that the number of trainable parameters of the convolutional base is 14,714,688, the next step was to freeze the base network using the code:

```
conv_base.trainable = False
```

In this way all the previous parameters were no more trainable.

At this point we added our model on top of the VGG16 base. We decided to use the last classifier used in the previous experiment, in the CNN from scratch, since it performed well in our training set. It consisted of two Dense layers and a Dropout layer to better generalize. The optimizer is always *Adam* and for its learning we decided to start with $10^{-}3$.

Before passing the input data in the convolutional base we decided to add a layer and use data augmentation to better avoid overfit. To have a clearer idea of the final model we report the code used:

```
inputs = keras.Input(shape=(180, 180, 3))
x = data_augmentation(inputs)
x = keras.applications.vgg16.preprocess_input(x)
x = conv_base(x)
x = layers.Flatten()(x)
x = layers.Dense(256, activation="relu")(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(18, activation="softmax")(x)

optimizer = keras.optimizers.Adam(learning_rate=1e-3)
```

```
11
12  model = keras.Model(inputs, outputs)
13  model.compile(loss='categorical_crossentropy',
14                optimizer=optimizer,
15                metrics=["accuracy"])
16
```

In the following figure 35 we can see the model summary of the model:

```
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 180, 180, 3)]     0

 sequential (Sequential)     (None, 180, 180, 3)       0

 tf.__operators__.getitem (S  (None, 180, 180, 3)      0
 licingOpLambda)

 tf.nn.bias_add (TFOpLambda)  (None, 180, 180, 3)      0

 vgg16 (Functional)          (None, 5, 5, 512)         14714688

 flatten (Flatten)           (None, 12800)             0

 dense (Dense)               (None, 256)               3277056

 dropout (Dropout)           (None, 256)               0

 dense_1 (Dense)             (None, 18)                4626

=================================================================
Total params: 17,996,370
Trainable params: 3,281,682
Non-trainable params: 14,714,688
_________________________________________________________________
```

Figure 35: model summary

At this point we started training the model using the callback *EarlyStopping* with patience 8 on the metric *validation loss*.

The plot of the training and validation accuracies and their losses are depicted in figure 36.

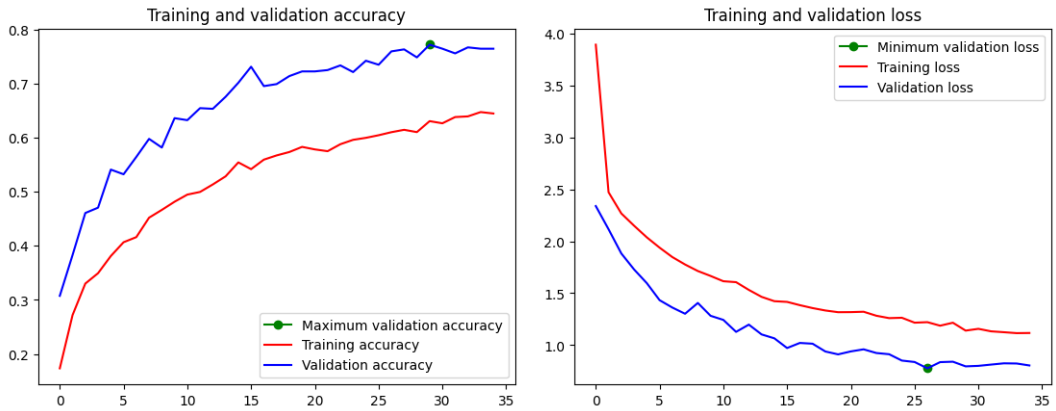


Figure 36: Training and validation accuracy and loss plots through the feature extraction phase

A particular aspect of these plots that immediately caught our attention is that the model performed better in the validation set than in the training set. This means that the model has a great generalization power, we hypothesize that the Dropout layer with a rate of 0.5 made a real difference.

VGG16 feature extraction		
Number of Epochs:	35	
	Loss	**Accuracy**
Train	1.1397	0.6303
Validation	0.7957	0.7716
Test	1.5926	0.6241

Table 9: Results achieved with the model

The results are pretty satisfactory. The validation accuracy is a lot greater than in the CNN from scratch, in fact in this experiment we reach a validation accuracy of **77%** and a validation loss of **0.80**, while in CNN from scratch the best value reached was **69%** for the validation accuracy and **1.24** for the validation loss. Also in the test set the model performs a lot better, in the CNN from scratch the best value was **59%** for the accuracy and **1.69** for the loss, while in this case the test accuracy reaches **62%** and the loss **1.59**.

6.1.2 VGG16 lower dropout rate and Fine Tuning last convolutional block

We proceed now with Fine Tuning using the same model as in Feature Extraction. The only difference in the model is that we decreased a bit the *dropout rate* to 0.25 (from 0.5) to see how this parameter affected the results since we noticed that the model performed better in the validation set than in the training set. For this experiment we followed the 5 steps for fine-tuning as explained at the beginning of Chapter 6, with the first 3 steps being exactly the same as in the case of feature extraction.

Firstly we froze all the layers in the convolutional base of VGG16 and then we added and trained our model on top of it for 100 epochs.

For the training we used the following keras callbacks:

```
callbacks_list = [
    keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=12),

    keras.callbacks.ModelCheckpoint(
    filepath=os.path.join(dir_name, 'VGG16_feature_extraction.h5'),
    monitor='val_loss',
    save_best_only=True,
    ),

    keras.callbacks.ReduceLROnPlateau(
    monitor='val_loss',
    factor=0.1,
    patience=5,
    )
]
```

EarlyStopping stopped the training after 66 epochs, the plot of the training and validation accuracies and their losses are depicted in figure 37.

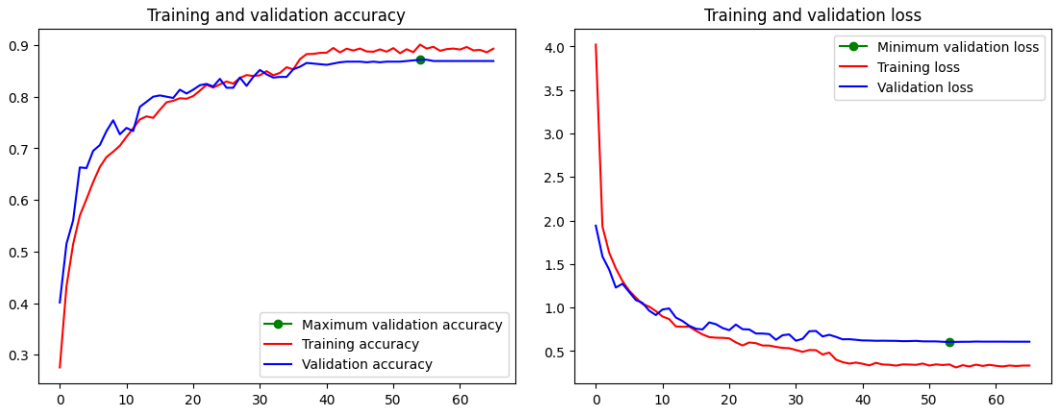


Figure 37: Training and validation accuracy and loss plots

We can see that these plots are different from the ones in fine-tuning, in this experiment the training and validation curves are a lot closer than before, and this is the effect of

the lower value of the *dropout rate*. After 37 epochs the *learning rate* of the optimizer decreased to 10^{-4} (from 10^{-3}) improving both the accuracy and loss of the model. From this epoch the performance of the model on the validation set is a bit worse than in the training set, which means that the model started a bit of overfit.

In the following table we report the model performance before fine-tuning:

VGG16 lower dropout rate feature extraction		
Number of Epochs:	66	
	Loss	**Accuracy**
Train	0.3122	0.9010
Validation	0.6060	0.8716

Table 10: Results achieved with the model

The lowered value of the *dropout rate* increased significantly the performance of the model! Now we reach a validation accuracy of **87%**, a lot better than the previous **77%**.

At this point we tried to further increase this performance with fine-tuning!

In general for fine-tuning it's better to use lower values of the learning rate than in the case of feature extraction, so we reduced the *learning rate* of the optimizer to 10^{-5}. The model performed already well in our dataset and we don't want to dramatically change the weights of the trainable layers, we want the model to adapt in small steps.

We decided to unfreeze the last 3 convolutional layers, which means that all layers up until 'block4_pool' should be frozen, and the layers 'block5_conv1', 'block5_conv2' and 'block5_conv3' should be trainable.

The code used to unfreeze is:

```
conv_base.trainable = True
set_trainable = False
for layer in conv_base.layers:
  if layer.name == 'block5_conv1':
    set_trainable = True
  if set_trainable:
    layer.trainable = True
  else:
    layer.trainable = False
```

We made sure to have unfrozen the right layers:

```
[68] # Make sure you have frozen the correct layers
     for i, layer in enumerate(conv_base.layers):
         print(i, layer.name, layer.trainable)
```

```
0 input_5 False
1 block1_conv1 False
2 block1_conv2 False
3 block1_pool False
4 block2_conv1 False
5 block2_conv2 False
6 block2_pool False
7 block3_conv1 False
8 block3_conv2 False
9 block3_conv3 False
10 block3_pool False
11 block4_conv1 False
12 block4_conv2 False
13 block4_conv3 False
14 block4_pool False
15 block5_conv1 True
16 block5_conv2 True
17 block5_conv3 True
18 block5_pool True
```

After this we jointly trained both these unfrozen layers with the part we added on top of the convolutional base.

EarlyStopping stopped the training after 24 epochs, the plot of the accuracies of training and validation and the plot of their losses are depicted in figure 38.

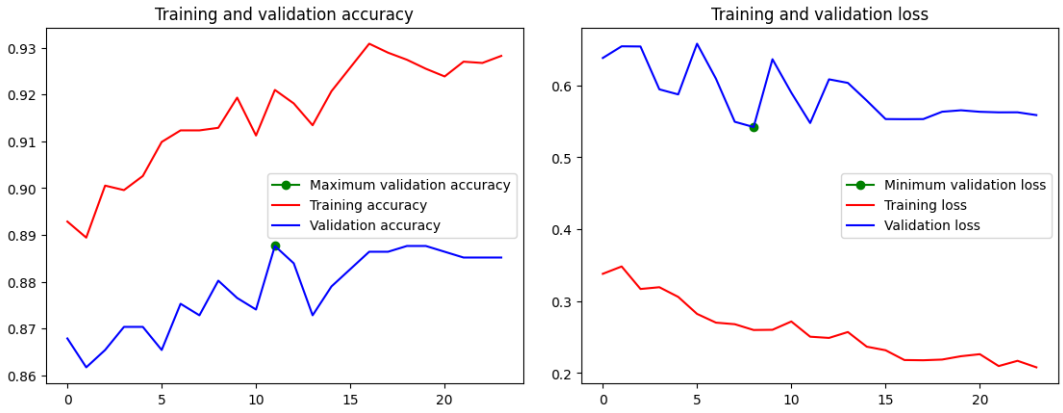


Figure 38: Training and validation accuracy and loss plots

Just looking at the plots we can see that fine-tuning made some improvements both in the accuracy and loss of training and validation curves.

In the following table we report the final model performance:

VGG16 with fine-tuning		
Number of Epochs:	24	
	Loss	**Accuracy**
Train	0.2187	0.9274
Validation	0.5633	0.8877
Test	2.4430	0.6727

Table 11: Results achieved with the fine-tuned model

From the results we notice that the validation accuracy improved from **87%** to nearly **89%**, and the validation loss decreased from **0.61** to **0.56**. The model performed quite well also in the test set reaching a test accuracy of **67%** and loss of **2.44**. We can conclude by saying that with feature extraction the model already performed well and fine-tuning was able to slightly improve it.

6.2 ResNet50V2

In this chapter we'll report the result obtained using another pre-trained CNN: ResNet50V2. The following picture shows its architecture:

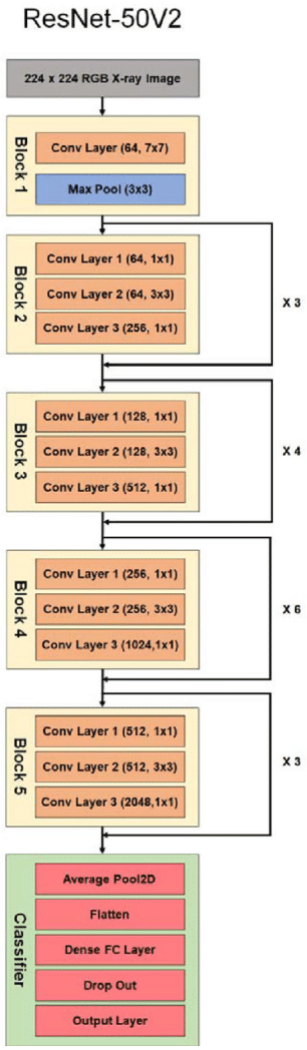


Figure 39: ResNet50V2 architecture

6.2.1 ResNet-50V2 Feature Extraction

In this experiment we followed the same steps as in VGG16 Feature extraction. Firstly we imported ResNet50V2 from Keras using the code:

```
from tensorflow.keras.applications import resnet_v2

resent_model = resnet_v2.ResNet50V2(
                include_top=False,
                weights='imagenet',
                input_shape=(180, 180, 3))
```

The values of parameters are the same as in VGG16. The argument *include_top=False* means not to include the densely connected classifier on top of the network. We will add our own classifier on top of the pre-trained base network. The argument *weights="imagenet"*

specifies the weight checkpoint from which to initialize the model, *input_shape=(180, 180, 3)* specifies the shape of the input tensors in the network.

In this way we obtained the resnet convolutional base, the number of its trainable weights is 23,519,360. At this point we proceeded to freeze the convolutional base and add our classifier on top of it. We decided to use the same model that we used for feature extraction with VGG16 but add a *MaxPooling2D* layer right after the convolutional base to create a down-sampled feature map, since the output from the convolutional base was pretty big.

We report the code used for the model to better understand:

```
inputs = keras.Input(shape=(180, 180, 3))
x = data_augmentation(inputs)
x = resnet_v2.preprocess_input(x)
x = resent_model(x)
x = layers.MaxPooling2D(pool_size=6)(x)
x = layers.Flatten()(x)
x = layers.Dense(256, activation="relu")(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(18, activation="softmax")(x)

optimizer = keras.optimizers.Adam(learning_rate=1e-3)

model = keras.Model(inputs, outputs)
model.compile(loss='categorical_crossentropy',
              optimizer=optimizer,
              metrics=["accuracy"])
```

The model summary is:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 180, 180, 3)]     0

sequential (Sequential)      (None, 180, 180, 3)       0

tf.math.truediv (TFOpLambda  (None, 180, 180, 3)       0
)

tf.math.subtract (TFOpLambd  (None, 180, 180, 3)       0
a)

resnet50v2 (Functional)      (None, 6, 6, 2048)        23564800

max_pooling2d_3 (MaxPooling  (None, 1, 1, 2048)        0
2D)

flatten (Flatten)            (None, 2048)              0

dense (Dense)                (None, 256)               524544

dropout (Dropout)            (None, 256)               0

dense_1 (Dense)              (None, 18)                4626

=================================================================
Total params: 24,093,970
Trainable params: 529,170
Non-trainable params: 23,564,800
_________________________________________________________________
```

EarlyStopping stopped the training after 33 epochs, the plot of the accuracies of training

and validation and the plot of their losses are depicted in figure 40.

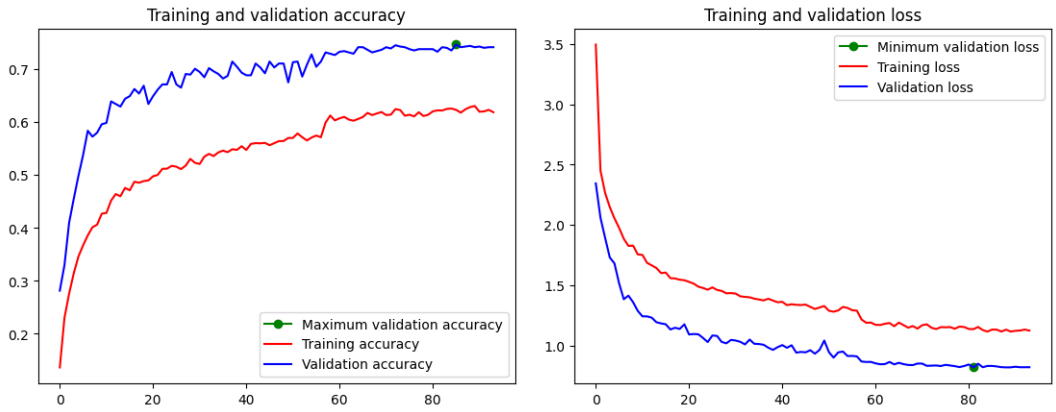


Figure 40: Training and validation accuracy and loss plots

In the following table we report the model performance with feature extraction:

ResNet feature extraction		
Number of Epochs: 92		
	Loss	**Accuracy**
Train	1.2410	0.5871
Validation	0.8963	0.7198

Table 12: Results achieved with the model

At this point we begin fine-tuning!

6.2.2 ResNet-50V2 Fine Tuning all the 3 blocks of conv 5

For fine-tuning we decided to unfreeze all the layers in conv 5, which means all the layers in "Block 5" if looking at the figure 39 showing ResNet50V2 architecture. In this way out of the 23,564,800 total parameters of the resnet convolutional base, 14,970,880 parameters became trainable and 8,593,920 remained non-trainable.

For the training, we used the same classifier and keras callbacks as in feature extraction but we decreased the learning rate of the optimizer to 10^{-4}.

The model summary is:

Layer (type)	Output Shape	Param #
input_2 (InputLayer)	[(None, 180, 180, 3)]	0
sequential (Sequential)	(None, 180, 180, 3)	0
tf.math.truediv (TFOpLambda)	(None, 180, 180, 3)	0
tf.math.subtract (TFOpLambda)	(None, 180, 180, 3)	0
resnet50v2 (Functional)	(None, 6, 6, 2048)	23564800
max_pooling2d_3 (MaxPooling2D)	(None, 1, 1, 2048)	0
flatten (Flatten)	(None, 2048)	0
dense (Dense)	(None, 256)	524544
dropout (Dropout)	(None, 256)	0
dense_1 (Dense)	(None, 18)	4626

Total params: 24,093,970
Trainable params: 15,500,050
Non-trainable params: 8,593,920

Figure 41: model used for fine-tuning

EarlyStopping stopped the training after 33 epochs, the plot of the accuracies of training and validation and the plot of their losses are depicted in figure 42.

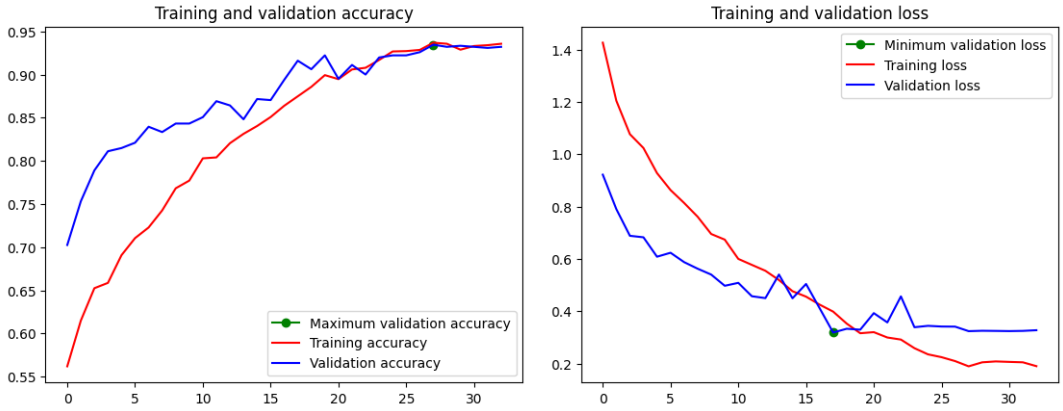


Figure 42: Training and validation accuracy and loss plots

We can see from the plots that the model increased its generalization power, in fact after 21 epochs the model performs better in the validation set than in the training set both looking at the accuracy and the loss plots.

In the following table we report the model performance with fine-tuning:

ResNet fine-tuning		
Number of Epochs: 33		
	Loss	**Accuracy**
Train	0.2084	0.9288
Validation	0.3250	0.9333
Test	1.4183	0.6897

Table 13: Results achieved with fine-tuning

Comparing the results with the ones in the table 12 we can see that with fine-tuning the validation accuracy increased from **72%** to **93%** and the validation loss decreased from **0.90** to **0.325**. Also the results in the test set are the best so far, reaching a test accuracy of **69%** and a test loss of **1.42**.

We can conclude that in this experiment fine-tuning the model made a great improvement in the overall performance!

6.2.3 ResNet with a less complex last layer

The last experiment done on the ResNet was to use a simpler block added at the end of the convolutional base.

```
inputs = keras.Input(shape=(180, 180, 3))
x = data_augmentation(inputs)
x = resnet_v2.preprocess_input(x)
x = resent_model(x)
x = layers.Flatten()(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(18, activation="softmax")(x)

optimizer = keras.optimizers.Adam(learning_rate=1e-2)

model = keras.Model(inputs, outputs)
model.compile(loss='categorical_crossentropy',
              optimizer=optimizer,
```

We've trained the model for 44 epochs (due to the early stopping) for the part of feature extraction reaching an accuracy of 88% and a validation accuracy of 84% as we can see in the figure 43.

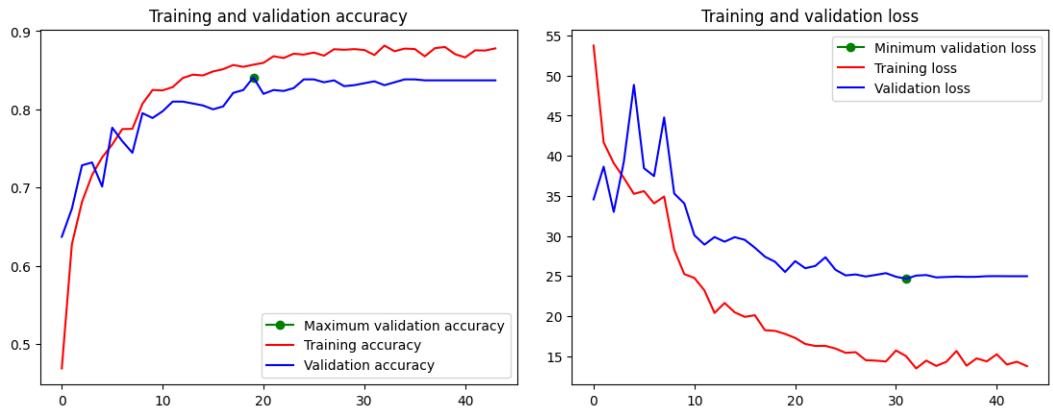


Figure 43: Training and validation accuracy and loss plots

After the feature extraction, that has reached a learning rate of 1e-5, we've started the fine tuning phase using the learning rate encountered at the end and unfreezing the layers from the 154 layer to the last one of the convolutional base (the entire 5th block).
The model was trained for 43 epochs and it has reached an accuracy of 96% and a validation accuracy of 92%.
The plots can be seen in figure 44.

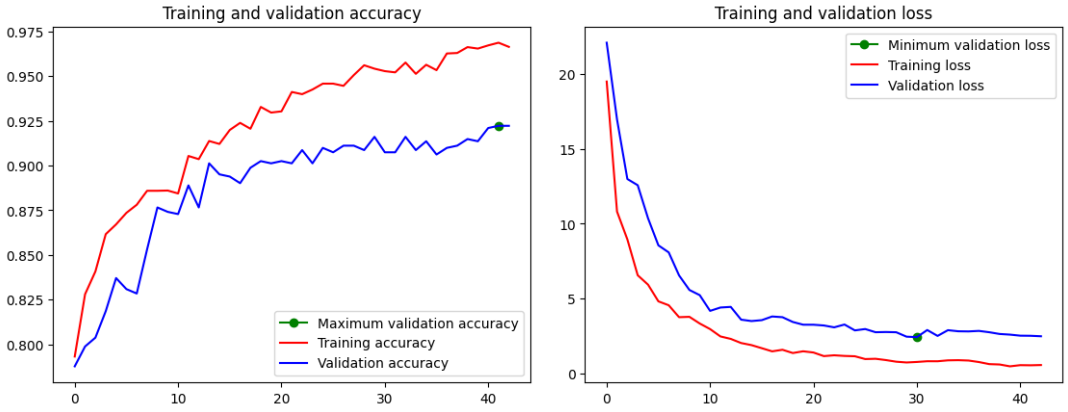


Figure 44: Training and validation accuracy and loss plots

This ResNet model is the best so far, so we've decided to conclude the experiments on the model due to the good results on the training and validation set observed; then we've performed an evaluation of the model expoloiting the test set obtaining the values exposed in table 14.

ResNet fine-tuning		
Number of Epochs: 43		
	Loss	**Accuracy**
Train	0.5526	0.9664
Validation	2.4771	0.9222
Test	13.5653	0.7212

Table 14: Results achieved with fine-tuning

The results of this model show how this model perform better on the test set, so with images never seen before, than the previous ones, despite a non significant reduction in the accuracy on the validation set (92% versus 93% obtained with the previous model).

6.3 Xception

In this chapter we'll report the result obtained using another pre-trained CNN: Xception. The following picture shows its architecture:

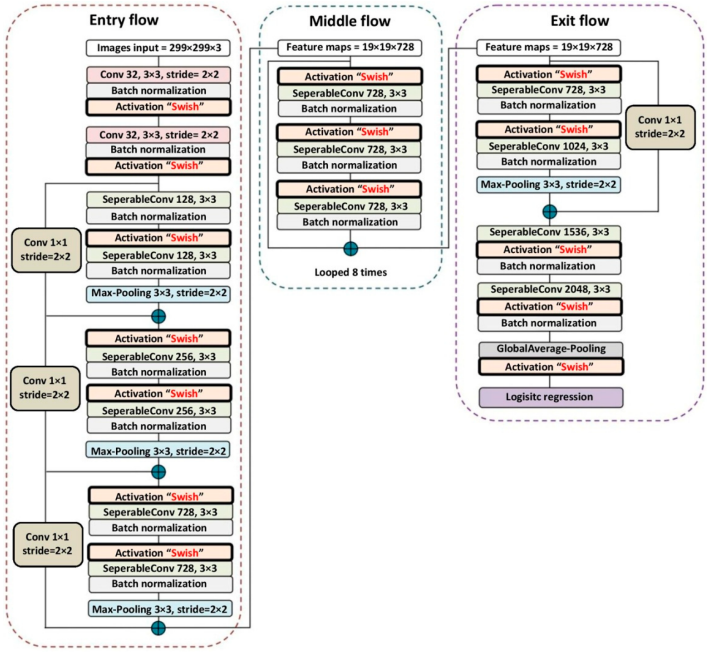


Figure 45: Xception architecture

6.3.1 Xception Feature Extraction

In this experiment we followed the same steps as in VGG16 and ResNet50V2 Feature extraction. Firstly we imported Xception from Keras using the code:

```
from tensorflow.keras.applications import xception

xception_base = xception.Xception(
    include_top=False,
    weights="imagenet",
    input_shape=(180, 180, 3)
)
```

In this way we obtained the Xception convolutional base, the number of its trainable weights is 20,806,952. At this point we proceeded to freeze the convolutional base and add our classifier on top of it. We decided to use the same model that we used for feature extraction with resnet and an initial learning rate for the optimizer of 10^{-3}.

We report the code used for the model to better understand:

```
inputs = keras.Input(shape=(180, 180, 3))
x = data_augmentation(inputs)
x = xception.preprocess_input(x)
x = xception_base(x)
x = layers.MaxPooling2D(pool_size=6)(x)
x = layers.Flatten()(x)
```

```
x = layers.Dense(256, activation="relu")(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(18, activation="softmax")(x)

optimizer = keras.optimizers.Adam(learning_rate=1e-3)

model = keras.Model(inputs, outputs)
model.compile(loss='categorical_crossentropy',
              optimizer=optimizer,
              metrics=["accuracy"])
```

The model summary is:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 180, 180, 3)]     0

sequential (Sequential)      (None, 180, 180, 3)       0

tf.math.truediv (TFOpLambda  (None, 180, 180, 3)       0
)

tf.math.subtract (TFOpLambd  (None, 180, 180, 3)       0
a)

xception (Functional)        (None, 6, 6, 2048)        20861480

max_pooling2d (MaxPooling2D  (None, 1, 1, 2048)        0
)

flatten (Flatten)            (None, 2048)              0

dense (Dense)                (None, 256)               524544

dropout (Dropout)            (None, 256)               0

dense_1 (Dense)              (None, 18)                4626

=================================================================
Total params: 21,390,650
Trainable params: 529,170
Non-trainable params: 20,861,480
```

EarlyStopping stopped the training after 90 epochs, the plot of the accuracies of training and validation and the plot of their losses are depicted in figure 46.

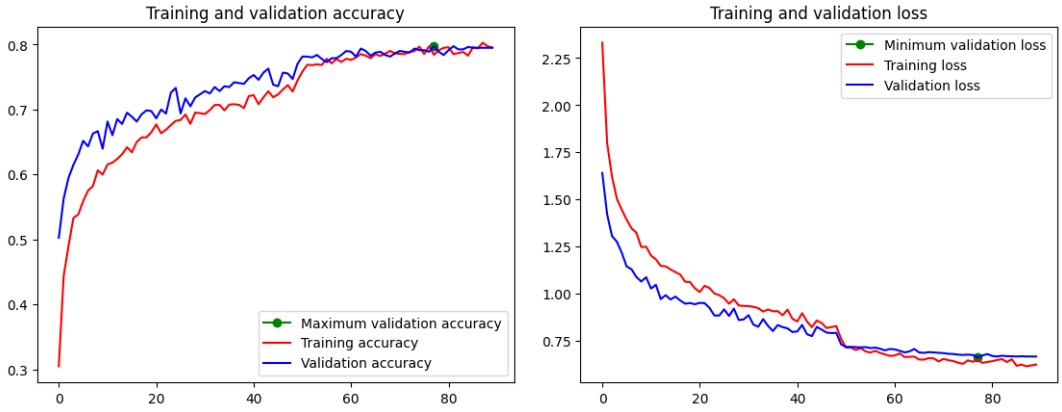


Figure 46: Training and validation accuracy and loss plots

In the following table we report the model performance with feature extraction:

Xception feature extraction		
Number of Epochs: 90		
	Loss	**Accuracy**
Train	0.6491	0.7842
Validation	0.6616	0.7975

Table 15: Results achieved with the model

At this point we've moved to the fine-tuning phase.

6.3.2 Xception Fine Tuning block 14

For fine-tuning we decided to unfreeze all the layers in the last block (block 14), so the last 6 layers. In this way out of the 20,861,480 total parameters of the resnet convolutional base, 4,748,800 parameters became trainable and 16,112,680 remains non-trainable.

For the training, we used the same classifier and keras callbacks as in feature extraction but we decreased the learning rate of the optimizer to 10^{-5}.

The model summary is:

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_4 (InputLayer)        [(None, 180, 180, 3)]     0

 sequential (Sequential)     (None, 180, 180, 3)       0

 tf.math.truediv (TFOpLambda  (None, 180, 180, 3)      0
 )

 tf.math.subtract (TFOpLambd  (None, 180, 180, 3)      0
 a)

 xception (Functional)       (None, 6, 6, 2048)        20861480

 max_pooling2d (MaxPooling2D  (None, 1, 1, 2048)       0
 )

 flatten (Flatten)           (None, 2048)              0

 dense (Dense)               (None, 256)               524544

 dropout (Dropout)           (None, 256)               0

 dense_1 (Dense)             (None, 18)                4626

=================================================================
Total params: 21,390,650
Trainable params: 5,277,970
Non-trainable params: 16,112,680
```

Figure 47: model used for fine-tuning

The plot of the accuracies of training and validation and the plot of their losses are depicted in figure 48.

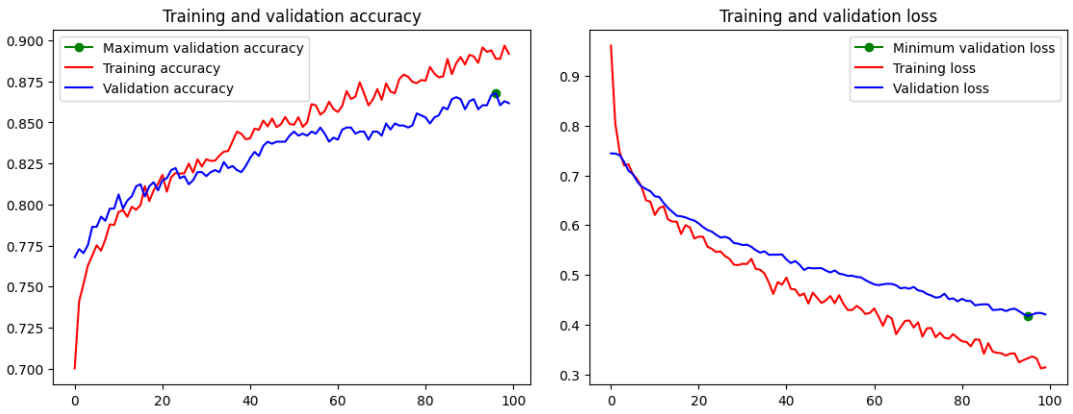


Figure 48: Training and validation accuracy and loss plots

In the following table we report the model performance with fine-tuning:

Xception fine-tuning		
Number of Epochs: 100		
	Loss	**Accuracy**
Train	0.3358	0.8889
Validation	0.4201	0.8679
Test	1.2452	0.6646

Table 16: Results achieved with fine-tuning

In this experiment fine-tuning the model made a great improvement in the overall performance.

6.3.3 Xception Fine Tuning from block 10 onwards

In this experiment for fine-tuning we decided to unfreeze all the layers from the block 10 onwards. In this way out of the 20,861,480 total parameters of the resnet convolutional base, 11,630,312 parameters became trainable and 9,231,168 remained non-trainable.

Also in this experiment we used for the training the same classifier and keras callbacks as before and a learning rate of the optimizer of 10^{-5}.

The model summary is:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_4 (InputLayer)         [(None, 180, 180, 3)]     0

sequential (Sequential)      (None, 180, 180, 3)       0

tf.math.truediv (TFOpLambda  (None, 180, 180, 3)       0
)

tf.math.subtract (TFOpLambd  (None, 180, 180, 3)       0
a)

xception (Functional)        (None, 6, 6, 2048)        20861480

max_pooling2d (MaxPooling2D  (None, 1, 1, 2048)        0
)

flatten (Flatten)            (None, 2048)              0

dense (Dense)                (None, 256)               524544

dropout (Dropout)            (None, 256)               0

dense_1 (Dense)              (None, 18)                4626

=================================================================
Total params: 21,390,650
Trainable params: 12,159,482
Non-trainable params: 9,231,168
_________________________________________________________________
```

Figure 49: model used for fine-tuning

The plot of the accuracies of training and validation and the plot of their losses are depicted in figure 50.

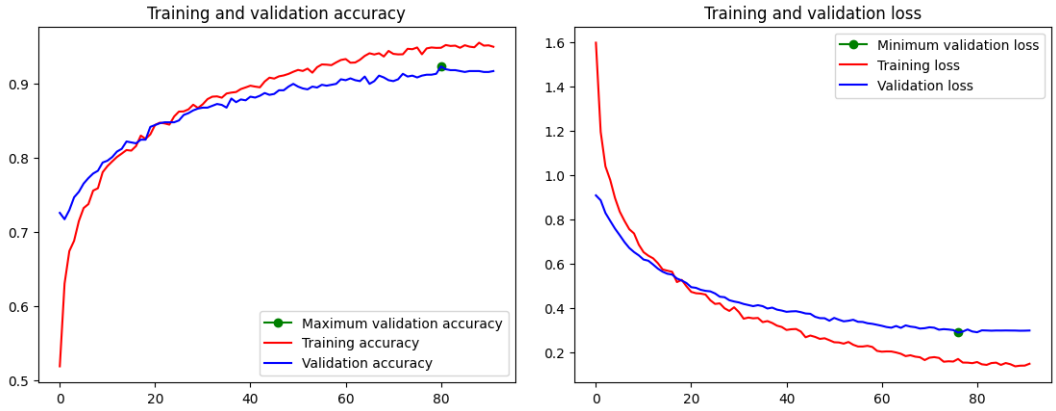


Figure 50: Training and validation accuracy and loss plots

In the following table we report the model performance with fine-tuning:

Xception fine-tuning		
Number of Epochs:	92	
	Loss	**Accuracy**
Train	0.1461	0.9524
Validation	0.2994	0.9198
Test	1.1725	0.7338

Table 17: Results achieved with fine-tuning

Also in this experiment fine-tuning the model made a great improvement in the overall performance, we've reached better accuracy and loss on training, validation and test set, compared to the first Xception model, as we can see comparing the table 16 and the table 17.

6.4 InceptionV3

In this chapter we'll report the results obtained using another pre-trained CNN: InceptionV3. The following picture shows its architecture:

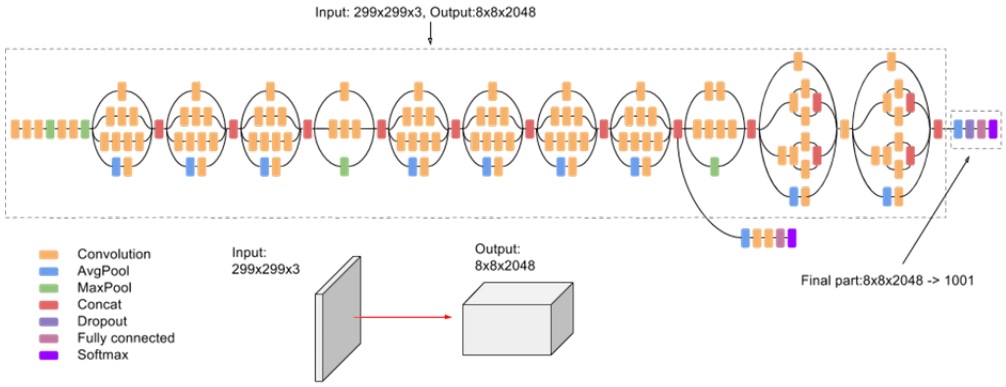


Figure 51: InceptionV3 architecture

6.4.1 InceptionV3 Feature Extraction

We followed the same steps as in the other pre-trained CNNs. Firstly we imported InceptionV3 from Keras using the code:

```
from keras import applications

base_model = applications.InceptionV3(weights='imagenet',
                                       include_top=False,
                                       input_shape=(180, 180,3))
```

In this way we obtained the InceptionV3 convolutional base, the number of its trainable weights is 21,768,352. At this point we proceeded to freeze the convolutional base and add our classifier on top of it. In this case we decided to use a simpler classifier consisting of only a Flatten layer, a Dropout layer and finally a Dense layer. The initial learning rate for the optimizer of 10^{-2}.

We report the code used for the model to better understand:

```
from keras.applications import inception_v3
inputs = keras.Input(shape=(180, 180, 3))
x = data_augmentation(inputs)
x = inception_v3.preprocess_input(x)
x = base_model(x)
x = layers.Flatten()(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(18, activation="softmax")(x)

optimizer = keras.optimizers.Adam(learning_rate=1e-2)

model = keras.Model(inputs, outputs)
model.compile(loss='categorical_crossentropy',
```

```
14                  optimizer=optimizer,
15                  metrics=["accuracy"])
```

The model summary is:

```
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 180, 180, 3)]     0

 sequential (Sequential)     (None, 180, 180, 3)       0

 tf.math.truediv (TFOpLambda  (None, 180, 180, 3)      0
 )

 tf.math.subtract (TFOpLambd  (None, 180, 180, 3)      0
 a)

 inception_v3 (Functional)   (None, 4, 4, 2048)        21802784

 flatten (Flatten)           (None, 32768)             0

 dropout (Dropout)           (None, 32768)             0

 dense (Dense)               (None, 18)                589842

=================================================================
Total params: 22,392,626
Trainable params: 589,842
Non-trainable params: 21,802,784
_________________________________________________________________
```

EarlyStopping stopped the training after 66 epochs, the plot of the accuracies of training and validation and the plot of their losses are depicted in figure 52.

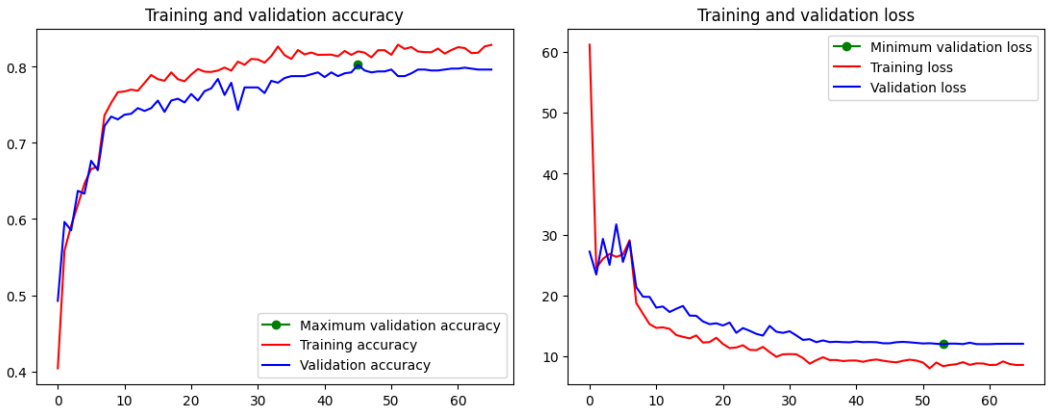


Figure 52: Training and validation accuracy and loss plots

In the following table we report the model performance with feature extraction:

InceptionV3 feature extraction		
Number of Epochs: 66		
	Loss	**Accuracy**
Train	8.6024	0.8246
Validation	12.0441	0.7988

Table 18: Results achieved with the model

6.4.2 InceptionV3 Fine Tuning

For fine-tuning we decided to unfreeze the last 42 layers. In this way out of the 22,392,626 total parameters of the inceptionv3 convolutional base, 6,909,650 parameters became trainable and 15,482,976 remained non-trainable.

For the training, we used the same classifier and keras callbacks as in feature extraction but we decreased the learning rate of the optimizer to 10^{-4}.

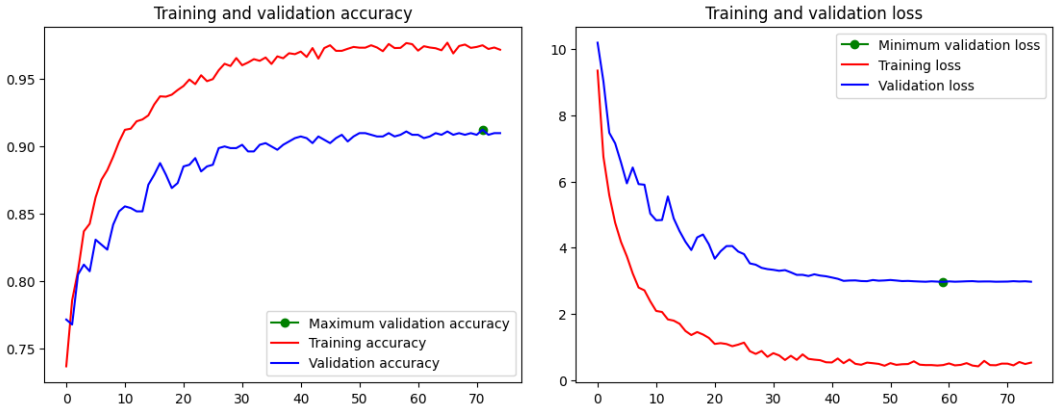


Figure 53: Training and validation accuracy and loss plots

In the following table we report the model performance with fine-tuning:

IncetionV3 fine-tuning		
Number of Epochs: 75		
	Loss	**Accuracy**
Train	0.4147	0.9770
Validation	2.9771	0.9111
Test	14.7087	0.7131

Table 19: Results achieved with fine-tuning

We can see that also this experiment fine-tuning the model made a great improvement with respect to feature extraction. We decided to do another experiment changing the

model since the results are good, but the loss is still much greater than in the other models that used different pre-trained CNN.

6.4.3 InceptionV3 fine tuning with another model

This experiment is pratically the same as the previous one with the only difference that we added a *max pooling* layer right after the inception base. We report the model summary for fine tuning:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_2 (InputLayer)         [(None, 180, 180, 3)]     0

sequential (Sequential)      (None, 180, 180, 3)       0

tf.math.truediv (TFOpLambda  (None, 180, 180, 3)       0
)

tf.math.subtract (TFOpLambd  (None, 180, 180, 3)       0
a)

inception_v3 (Functional)    (None, 4, 4, 2048)        21802784

max_pooling2d_4 (MaxPooling  (None, 1, 1, 2048)        0
2D)

flatten (Flatten)            (None, 2048)              0

dropout (Dropout)            (None, 2048)              0

dense (Dense)                (None, 18)                36882

=================================================================
Total params: 21,839,666
Trainable params: 36,882
Non-trainable params: 21,802,784
_________________________________________________________________
```

We report only the results of the fine-tuning that was performed in the same way as before.

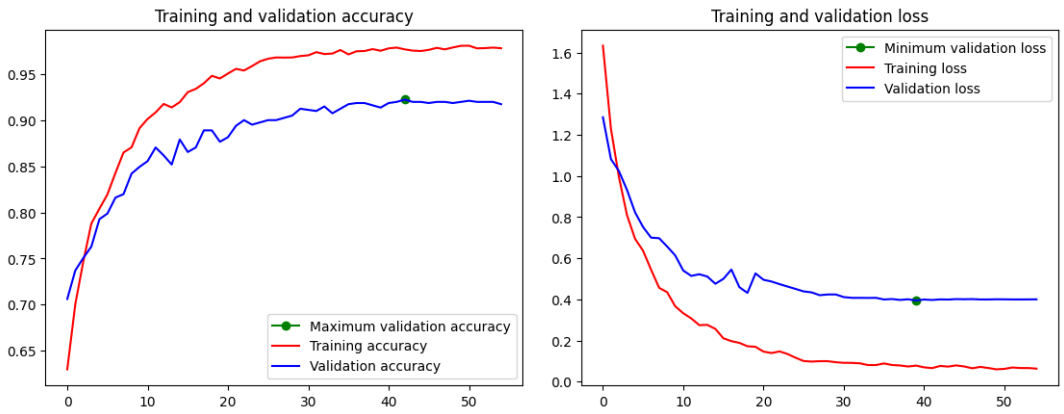


Figure 54: Training and validation accuracy and loss plots

IncetionV3 fine-tuning		
Number of Epochs: 55		
	Loss	**Accuracy**
Train	0.0656	0.9786
Validation	0.3969	0.9198
Test	1.6664	0.7221

Table 20: Results achieved with fine-tuning

While the accuracies are more or less the same as the first fine tuning, the losses decreased a lot: from **14.71** to **1.67** for the test set, from **2.98** to **0.40** for the validation set.
With this model we have concluded the experiments regarding the pre-trained models.

6.5 Comparing the results of the pre-trained models

So far we've fine-tuned four different pre-trained models obtaining good accuraces on both the training and validation set. These models must perform predictions on images never seen during the training phase, the result of these predictions can be seen in the previous tables in the row named test, that we've put together in the following table.

Results of the predictions		
	Loss	**Accuracy**
VGG16	2.4430	0.6727
ResNet	13.5653	0.7212
Xception	1.1725	0.7338
InceptionV3	1.6664	0.7221

Table 21: Best results of the prediction using the test set

The accuracy in this table is just a summary of the accuracies of the predictions on the different classes, therefore we've built the confusion matrix for each model to analyze more in detail which classes are less recognized.

The next four figures contain the confusion matrix and a table with the accuracies obtained on the test set for each model present in table 21.

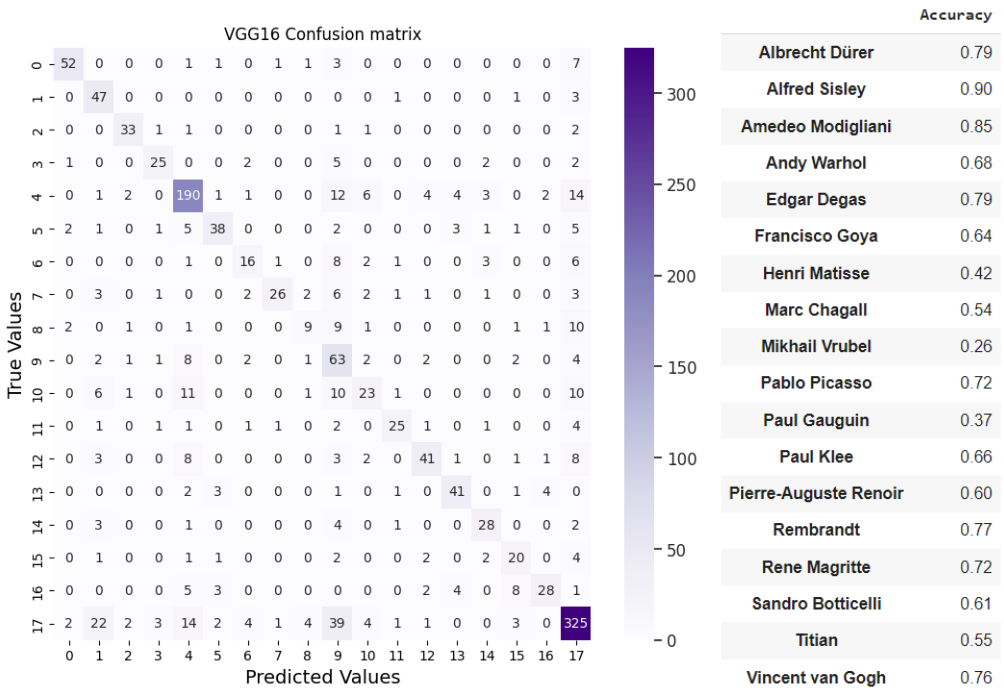


Figure 55: Confusion matrix and accuracies on the test set of the VGG16-based model

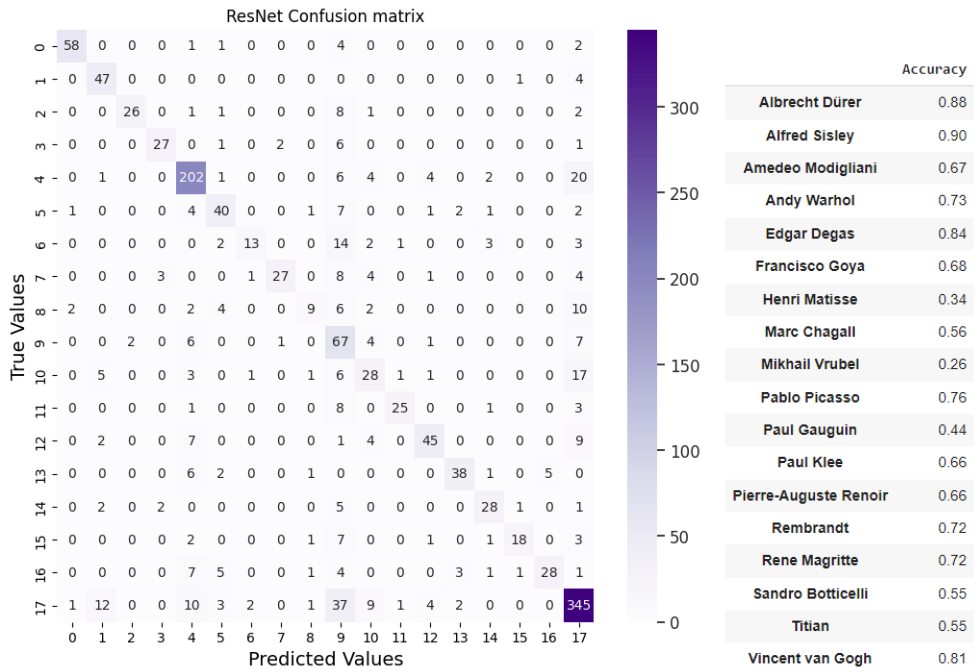


Figure 56: Confusion matrix and accuracies on the test set of the ResNet-based model

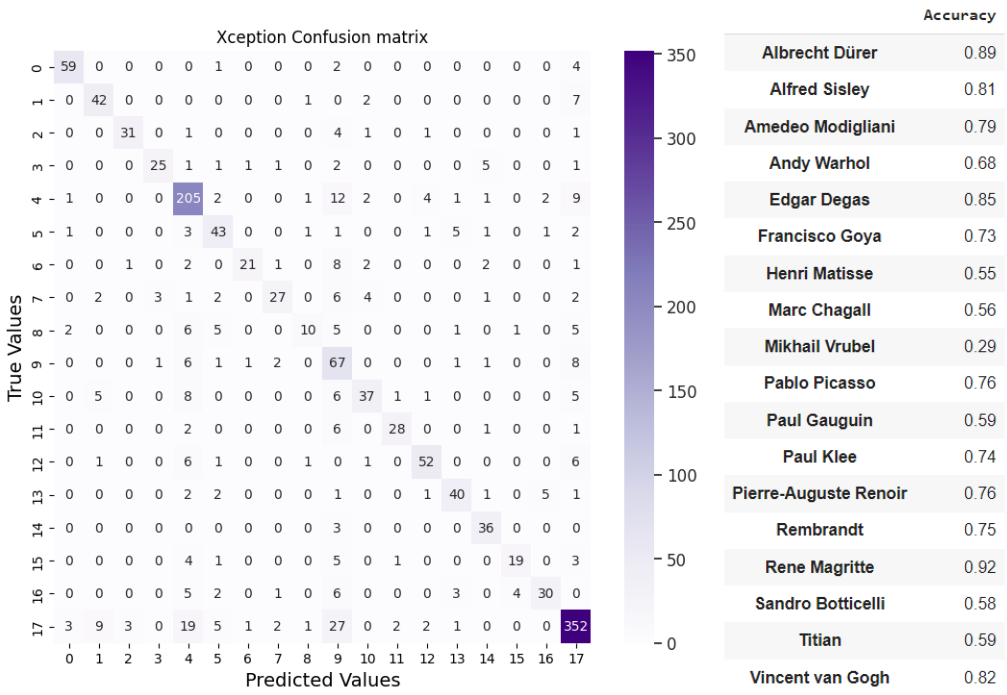


Figure 57: Confusion matrix and accuracies on the test set of the Xception-based model

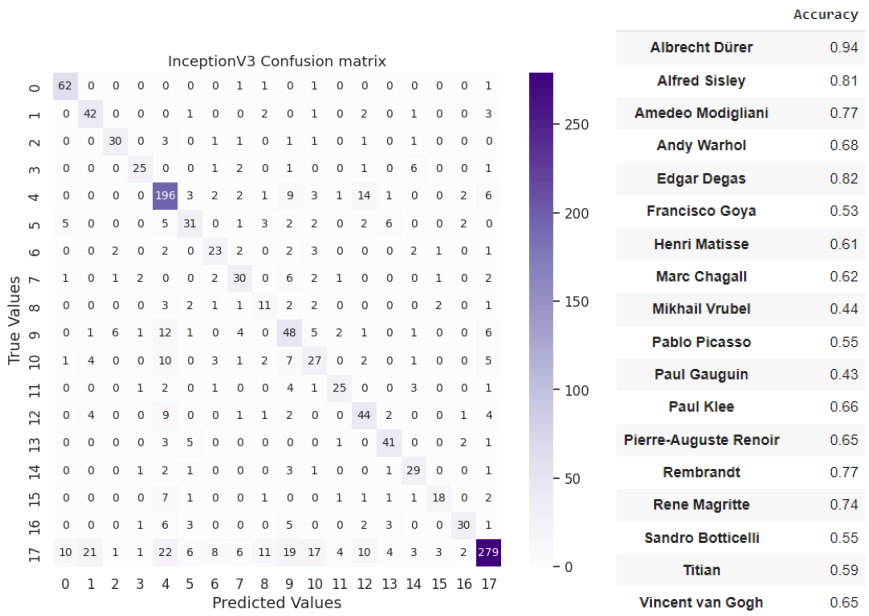


Figure 58: Confusion matrix and accuracies on the test set of the InceptionV3-based model

Observing the different confusion matrices it is possible to see that there are some classes that have a bad accuracy on each model:

- *Mikhall Vrubel*
- *Paul Gauguin*
- *Henri Matisse*

Taking as example the ResNet and its prediction (figure 56) regarding *Mikhall Vrubel* (class 8) we have seen that the accuracy is **0.26**, in fact the majority of predictions assigned *Vincent Van Gogh* as class. Another artist that is recognized poorly is *Paul Gauguin* (Class 10), also in this case there is a misclassification with *Vincent Van Gogh*, but not for the majority of samples.
We've tried to find the reasons behind this behaviour: Vrubel is the artist with the less number of paintings in our dataset (figure 5) and combined to the fact that its paintings are very different from each other, it is possible that the data contained in the test set are very distant from the paintings used during the training phase.
Starting with this assumption we've tried different methods to increase the accuracy of the models. The first method was to try to train the model on the original training set without data augmentation and exploiting the class weights; in this way we can impose more attention to the classes with less paintings. This method was applied to the InceptionV3 model and unfortunately it doesn't solve the problem at all, on the contrary it introduce the problem of the overfitting never encountered so far using pre trained models, as we can see in figure 59.

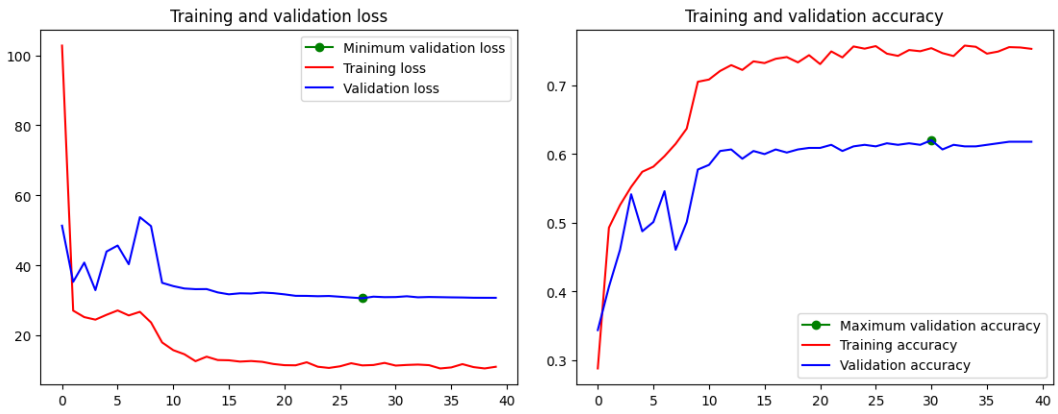


Figure 59: Plot of the loss and accuracy of the InceptionV3 model trained on the starting dataset with class weights

The second method that we've used was to move from the test set to the training set some samples of the classes that were misclassified the most (the three artists in the previous list).
We've tested this method on the ResNet model, to see if this approach would have led to benefits.

6.5.1 Retrieving the misclassified paintings

In order to select the images to move we've defined the following functions:

```
def get_images_predictions(test_dataset, true_class, predicted_class):

  images = []

  # iterate over the dataset
  for image_batch, label_batch in test_dataset:
    # compute predictions
    preds = model.predict(image_batch)

    # retrieve the misclassified images
    for i in range(len(label_batch)):
      if((label_batch[i][true_class] == 1) and
      (np.argmax(preds, axis=-1)[i] == predicted_class)):
          images.append(image_batch[i].numpy().astype("uint8"))

  return images


  def print_predictions(images, true_class, predicted_class):
  fig = plt.figure(figsize=(9.9, 9.8))
  fig.suptitle("Predicted class: " + class_names[predicted_class]+
  "\nCorrect class: "+class_names[true_class])

```

```
    for i, image in enumerate(images):
      ax = plt.subplot(4,(len(images)//4)+1,i + 1)
      plt.imshow(images[i])
      ax.axis("off")

    plt.tight_layout()
    plt.show()
```

The first is used to retrieve from the batch of images the ones that are misclassified as the "predicted_class", while the second is used to print the images.
Using these function we've retrieved the images of:

- Paul Gauguin predicted as Vincent Van Gogh
- Vrubel predicted as Vincent Van Gogh
- Henri Matisse predicted as Pablo Picasso

Then we've selected a subset of each set of images and moved them to the training set (only a subset because, if we move all the images to the training set, becomes impossible to evaluate if this method gives improvements since the misclassified images would no longer be present in the test set). The sets from which we've selected the images are presented in the following figures.

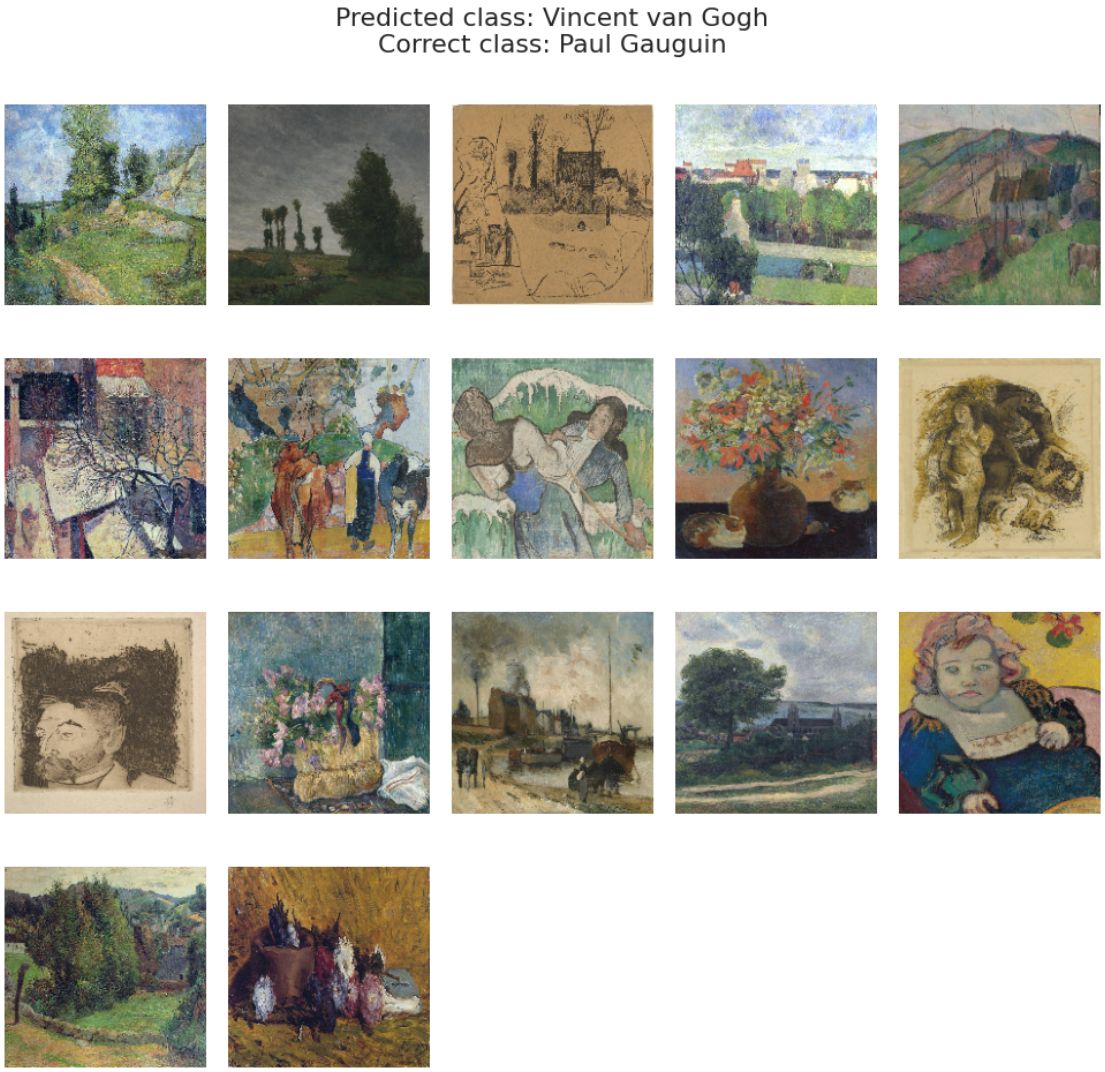


Figure 60: Images predicted as Van Gogh despite belonging to Gauguin

As we can see these Paul Gauguin's paintings can be easily misunderstood since they appear very similar to the style used by Vincent Van Gogh, in fact they are two artists representative of the Post Impressionism and, more important, they lived together in Arles during the 1888 influencing each others.

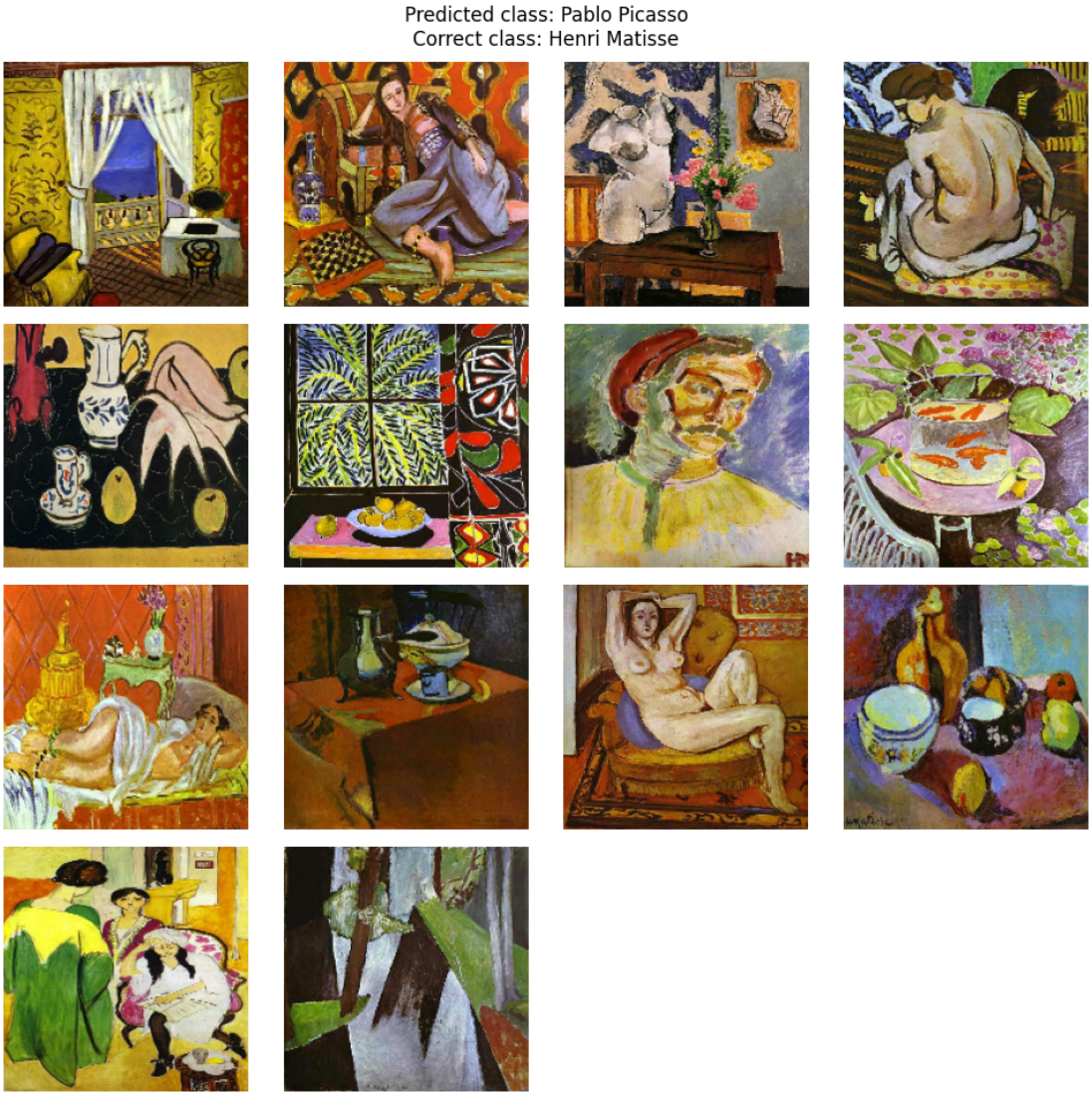


Figure 61: Images predicted as Picasso despite belonging to Matisse

In the paintings in figure 61 we can see the ones belonging to Henri Matisse misclassified as Picasso, but also in this case we've the same situation seen before: due to its rivalry they were influenced by each other.
The following are the images assigned to Van Gogh while they belong to Vrubel.

Figure 62: Images predicted as Van Gogh despite belonging to Vrubel

6.5.2 Training ResNet with the new dataset

We used the ResNet model in which we obtained the best accuracy on the test set (see the following snippet of code).

```
inputs = keras.Input(shape=(180, 180, 3))
x = data_augmentation(inputs)
x = resnet_v2.preprocess_input(x)
x = resent_model(x)
x = layers.Flatten()(x)
x = layers.Dropout(0.5)(x)
outputs = layers.Dense(18, activation="softmax")(x)

optimizer = keras.optimizers.Adam(learning_rate=1e-2)

model = keras.Model(inputs, outputs)
model.compile(loss='categorical_crossentropy',
```

```
13                  optimizer=optimizer,
14                  metrics=["accuracy"])
15
```

We've used the same approach used for the last ResNet trained so far (same last layer, parameters and unfreezed layers); after the feature extraction we've reached 87% of training accuracy and 85% of validation accuracy (figure 63) and after the fine tuning process we've obtained a training accuracy of 96% and a validation accuracy of 92% (figure 64).

Figure 63: Plot of the loss and accuracy of the ResNet after the feature extraction

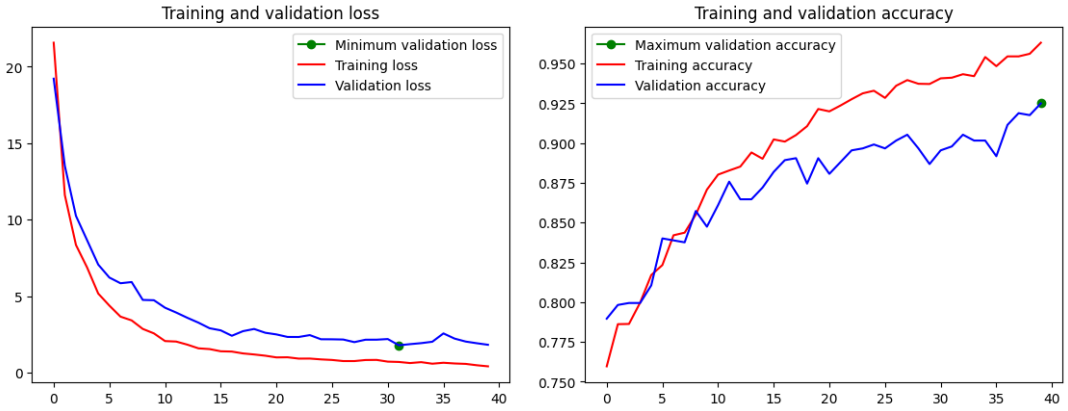


Figure 64: Plot of the loss and accuracy of the ResNet after the fine tuning

The result on the test set are the following:

Results of the prediction		
	Loss	**Accuracy**
ResNet	6.7302	0.7606

Table 22: Results of the prediction using the test set

Now we can compare the results of the ResNet trained on the different training sets:

Results of the predictions		
	Loss	**Accuracy**
ResNet with balanced dataset	13.5653	0.7212
ResNet with more samples	6.7302	0.7606

Table 23: Comparison between the two ResNet models

As we can see the new dataset improves the accuracy and reduces the loss of the model, when it is tested against images never seen. More in detail we can compare the accuaracies and the confusion matrices of the two models, since we have to see if the accuracies of the classes of which we've moved the images from the test set to the training set are better:

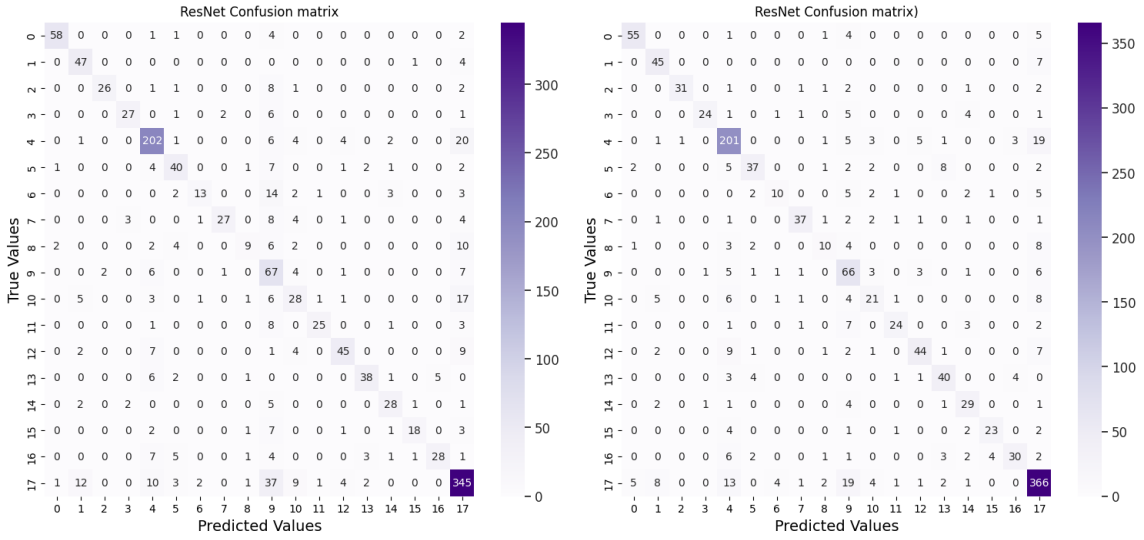


Figure 65: In order, the confusion matrix of the ResNet model with the balanced training set and the confusion matrix of the ResNet with more samples of the minority classes.

	Accuracy
Albrecht Dürer	0.88
Alfred Sisley	0.90
Amedeo Modigliani	0.67
Andy Warhol	0.73
Edgar Degas	0.84
Francisco Goya	0.68
Henri Matisse	0.34
Marc Chagall	0.56
Mikhail Vrubel	0.26
Pablo Picasso	0.76
Paul Gauguin	0.44
Paul Klee	0.66
Pierre-Auguste Renoir	0.66
Rembrandt	0.72
Rene Magritte	0.72
Sandro Botticelli	0.55
Titian	0.55
Vincent van Gogh	0.81

	Accuracy
Albrecht Dürer	0.83
Alfred Sisley	0.87
Amedeo Modigliani	0.79
Andy Warhol	0.65
Edgar Degas	0.84
Francisco Goya	0.63
Henri Matisse	0.36
Marc Chagall	0.77
Mikhail Vrubel	0.36
Pablo Picasso	0.75
Paul Gauguin	0.45
Paul Klee	0.63
Pierre-Auguste Renoir	0.65
Rembrandt	0.75
Rene Magritte	0.74
Sandro Botticelli	0.70
Titian	0.59
Vincent van Gogh	0.86

Figure 66: Accuracies on the test set of the ResNet model with the balanced training set and the ones of the ResNet with more samples of the minority classes.

The results are not good as we expected: the training of the model with the different dataset changed the accuracies of all the classes, reducing some classes well recognized before and increasing some others, but the accuracies of the classes of which we've moved the images from the test to the training set to increase the samples during the training doesn't have any improvement, only Vrubel has an increase of 0.10, while Gauguin and Matisse have the same accuracies as before.
In conclusion we don't have observed any significant improvement using this method, probably, due to the fact that we don't have enough samples of the classes, and it is hard to find new images since in the context in which we're operating the images that we can use are limited: we've already all the paintings available for each artist.

7 Ensemble

Ensemble involves combining the predictions of multiple neural network models to produce a more accurate and robust prediction, improving the performance of the individual models.

There are different types of ensemble learning methods in deep neural models, such as bagging, boosting, and stacking. Bagging for example involves training multiple models independently on different random subsets of the training data (bootstrap sampling), and then combining their predictions using an average or majority vote.

In our project we tried to combine the predictions of our models in a way similar to bagging but without using bootstrap sampling. Unfortunately, for a matter of time, we decided to use the best models we saved before for the ensemble, and all of them were trained using the same training set. The idea behind training the models with different subsets of the training data is that the models will have a slightly different perspective of the data and will make different errors, combining the predictions the errors will cancel each other out, resulting in a more accurate prediction. This is why for these experiments we already expected that the final performance of the ensemble could be suboptimal, but still expected an improvement in the overall performance with respect to the individual models.

We decided to use 4 models for the ensemble, one for each pre-trained CNN. We chose the models that performed best in the previous tests, we reported their performance in table 21.

7.1 Ensemble technique: Mean averaging

Averaging means taking the mean of the individual models' predictions as the ensemble model's prediction. To better understand our code let's remember that the output of our models is an array of probabilities, where each element of the array represents the probability of the corresponding class. The predicted class is the class with the highest probability in the output array. In the code we didn't exactly compute the mean of the probabilities of the individual models but only the sum of these probabilities since the final predicted class is the same.

We report the code of this computation:

```
def ensamble_avg_predict(models):
  y_preds_models=[None]*len(models)
  preds_models=[None]*len(models)
  y_true = []  # store true labels
  y_pred = []  # store ensamble predicted labels

  # iterate over the dataset
  for image_batch, label_batch in test_dataset:
    # append true labels
    y_true.append(label_batch)

    # compute predictions
    for i in range(len(models)):
```

```
        preds_models[i]=models[i].predict(image_batch)

    # compute sum of predicted labels of the models
    preds = np.array(list(map(sum, zip(*preds_models))))

    # append predicted labels of the ensemble model
    y_pred.append(np.argmax(preds, axis = - 1))

# convert the true and predicted labels into tensors
correct_labels = tf.concat([item for item in y_true], axis = 0)
predicted_labels = tf.concat([item for item in y_pred], axis = 0)

return correct_labels, predicted_labels
```

The above function takes as input the list of the models for the ensemble and produces as output the tensors of the true labels and the predicted labels of the ensemble model on the test set. With these output tensors we were able to plot the Confusion Matrix and a table containing the accuracies reached for each artist.

The Confusion Matrix is depicted in figure 67:

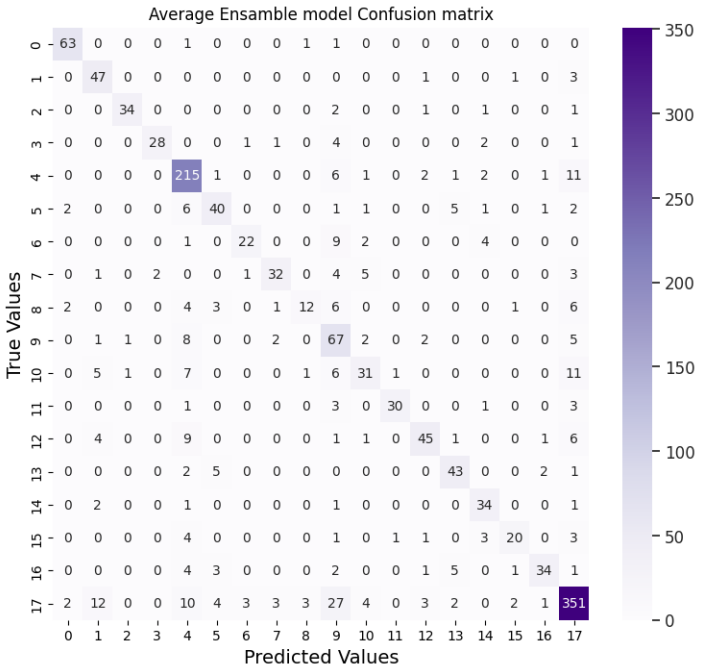


Figure 67: Confusion Matrix of the average ensemble model

Looking at the Confusion Matrix we can already notice how the majority of the labels were predicted correctly (the numbers in the diagonal are the true positives). Looking at a row, the number of false positives is the sum of all the values in the raw without considering the single box with the number of true positives. We can observe how the ensemble model makes similar errors to the individual models, it also tends to predict the majority classes in the false positives. For example looking at class 10 (corresponding to the artist *Paul Gauguin*), 11 images were erroneously assigned to class 17 (*Vincent van Gogh*) and 7 to class 4 (*Edgar Degas*), which are the majority classes.

In figure 68 are reported the accuracies obtained for each class:

	Accuracy
Albrecht Dürer	0.95
Alfred Sisley	0.90
Amedeo Modigliani	0.87
Andy Warhol	0.76
Edgar Degas	0.90
Francisco Goya	0.68
Henri Matisse	0.58
Marc Chagall	0.67
Mikhail Vrubel	0.34
Pablo Picasso	0.76
Paul Gauguin	0.49
Paul Klee	0.79
Pierre-Auguste Renoir	0.66
Rembrandt	0.81
Rene Magritte	0.87
Sandro Botticelli	0.61
Titian	0.67
Vincent van Gogh	0.82

Figure 68: Accuracies per class

We can see how the ensemble model performs pretty well exceeding **60%** of accuracy for all classes with the only exceptions of *Mikhail Vrubel* and *Paul Gauguin*. In particular the performance for Vrubel is only of **34%** and it drastically lowers the overall performance. The reason is the same as we said before, the errors of the individual models weren't much canceled and the models with VGG16, with ResNet, and with Xception performed very poorly in this class (respectively 26%, 26% and 44% of accuracy). However, despite this, the ensemble model reached an overall accuracy of **74%**, and comparing this performance with the ones reported in table 21 we can see how the performance of the ensemble model is greater than all the ones of the individual models!

7.2 Ensemble technique: Majority vote

Voting means choosing the most common prediction (the mode) of the predicted values of the ensembled models.

We report the code of this computation:

```
def ensamble_mode_predict(models):
  y_preds_models=[None]*len(models)
  preds_models=[None]*len(models)
  batch_preds=[None]*len(models)
  y_true = []  # store true labels
```

```
    y_pred = []  # store ensamble predicted labels

    # iterate over the dataset
    for image_batch, label_batch in test_dataset:
      # append true labels
      y_true.append(label_batch)

      # compute predictions
      for i in range(len(models)):
        preds_models[i] = models[i].predict(image_batch)
        batch_preds[i] = np.argmax(preds_models[i], axis = - 1)

      # compute the mode of the predictions
      m = stats.mode(batch_preds, keepdims=True)
      mode_preds= m[0][0]

      # append predicted labels of the ensemble model
      y_pred.append(mode_preds)

    # convert the true and predicted labels into tensors
    correct_labels = tf.concat([item for item in y_true], axis = 0)
    predicted_labels = tf.concat([item for item in y_pred], axis = 0)

    return correct_labels, predicted_labels
```

The Confusion Matrix of this ensemble model is depicted in figure 69:

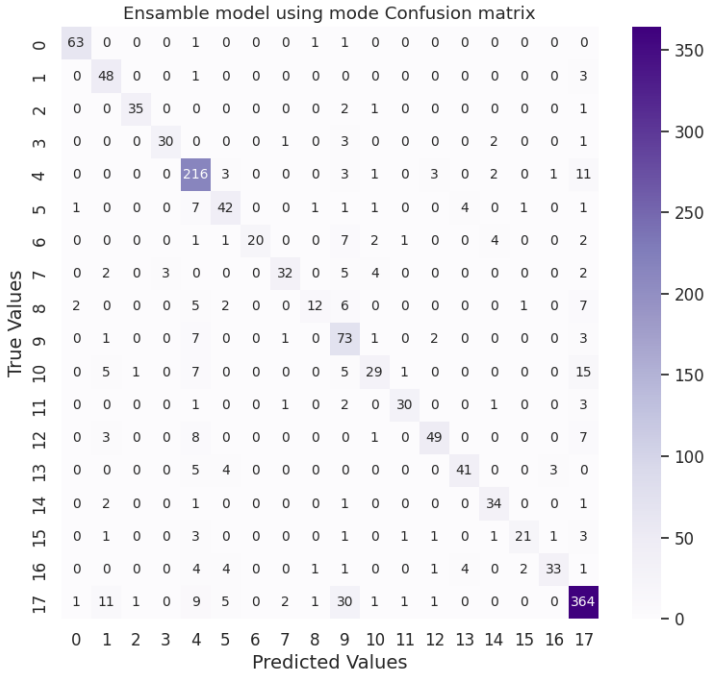


Figure 69: Confusion Matrix of the majority vote ensemble model

The Confusion Matrix obtained is pretty similar to the previous ensemble model that

used the mean. All the observations made before are still valid.

In figure 70 are reported the accuracies obtained for each class:

	Accuracy
Albrecht Dürer	0.97
Alfred Sisley	0.92
Amedeo Modigliani	0.87
Andy Warhol	0.78
Edgar Degas	0.88
Francisco Goya	0.75
Henri Matisse	0.55
Marc Chagall	0.65
Mikhail Vrubel	0.29
Pablo Picasso	0.78
Paul Gauguin	0.44
Paul Klee	0.74
Pierre-Auguste Renoir	0.72
Rembrandt	0.75
Rene Magritte	0.77
Sandro Botticelli	0.55
Titian	0.57
Vincent van Gogh	0.79

Figure 70: Accuracies per class

Also in this case the ensemble model performs worse in classifying the images of *Mikhail Vrubel* and *Paul Gauguin*. The overall performance is of **71%** which is still pretty good but a bit worse than the ensemble model that used the mean of the predictions. The reason is probably that in many predictions we didn't have a majority vote but 2 models voted for one class and the other 2 for another one. The final prediction is one of the two classes chosen randomly without considering their probabilities.

8 Conclusions and feasible future improvements

In this report we analyzed different deep-learning techniques to solve the task of artist recognition.

We started with a simple baseline CNN model from scratch and made sure that the dataset preprocessed so that all its classes were balanced (using *Data Augmentation*) performed in general a lot better than the unbalanced dataset with *weights to classes*.

With the balanced dataset we then proceeded in deploying many CNN models from scratch to improve the accuracy and loss metrics, in particular the main problem was that the model overfitted our training data. We tried many techniques to fight it, we changed many architectures of the models, added *dropout* layers, and used *L1 and L2 weight regularization* with three Keras callbacks. In general these techniques improved the models, we saw that models with more than 2 million trainable parameters performed worse. The best model so far was the one reported in table 7, it contains a Dropout layer, L2 regularization and reached a validation accuracy of 69% and a test accuracy of 59%.

We applied two visualization techniques to one model to better understand how it transforms an input image to recognize the class, interpreting the visualizations of intermediate activations and the heatmaps.

Then in the second part of our report we used 4 pre-trained CNN: VGG16, ResNet50V2, Xception, and InceptionV3. We noticed that all of them performed a lot better than the CNN from scratch models, exceeding 88% accuracy in the validation set and 67% in the test set. In all of them fine-tuning performed always better than feature extraction. The best-performing model was the one using RenNet50V2 which reached an accuracy of 92% on the validation set and 72% on the test set.

We compared the results of the pre-trained models plotting their Confusion Matrix and the tables with the accuracies per class. We saw that the models had some difficulty in predicting the classes with fewer samples (*Mikhall Vrubel*, *Paul Gauguin* and *Henri Matisse*) and to solve this problem we tried to add more samples of these classes in the training set. However the results weren't as good as expected because these classes remained the most misclassified ones. This is probably because we didn't have enough samples of the classes, but we think that adding them from another dataset could improve their performance.

Finally we combined the four best-performing pre-trained models in an Ensemble model, using Mean Averaging and Majority vote techniques. Both of them performed quite well but the Ensemble model with Mean averaging improved the performance of the individual models and reached an overall accuracy on the test set of 74%. This is the best model we developed so far for our task and its final performance is pretty satisfactory. However we didn't employ all the different ensemble techniques so it is possible we could have achieved even better results.

A possible improvement is to find manually the paintings not present in the dataset for the minority classes, but it is a task that is too difficult. Another improvement can be to train different models on different portions of the dataset and use the ensemble of them to build the result. With the hyperparameter optimization we can improve the accuracies of the models built from scratch. Finally another path that we can use is to move towards

different methods like using Genetic Algorithms to test if these kinds of algorithms will work better than the ones used during our experiments.