Painter style recognition

Machine Learning

Final Project

Caravaggio

Manet

Van Gogh

GROUP

Vito Turi	0622701795
Ferdinando Sica	0622701794
Camilla Spingola	0622701698
Mattia Marseglia	0622701697

L.M. Computer Engineering

A.A. 2021/2022

Summary

1. Project Introduction

The main topic in which our project is inserted in the image classification, one of the most common and important machine learning problem. Artistic creation is amongst the highest forms of human expression and imagination. The ability to communicate our vision sets us apart from all other beings. Painting, being an expression of visual language, has attracted and connected brilliant human minds since the beginning of civilization. After a long way, a stage has finally been reached where not only humans but also computers, another brilliant creation of human minds, are creating paintings. With its efficiency in identifying patterns, the human brain can quickly identify abstract concepts such as style, trait, and intensity, which are as related and unique to the body of the artist's work as a kind of personal signature. Artificial neural networks are mathematical structures that can learn and later recognize patterns. In this way, this project aims to use convolutional neural networks as a technique for identifying and classifying the authorship of artists in different images of their paintings. This report is structured as follows. In this first section we describe the characteristics of our project in terms of requirements and scope. In the second section we present the architecture of the project, so the dataset, the model and the used learning techniques. In the third section we analyze the learning procedure for the network. Finally, in the fourth section, we discuss the conclusions.

Goals

The aim of this project is to recognize photos of paints of three different painters. In particular:

Functional requirements

- Given an image containing a picture of: Caravaggio, Manet or Van Gogh, the network must be as much as possible able to recognize who is the author.
    - The image could contain different objects.
    - The image could contain different poses.
    - The image could contain different genres.
    - The image could contain different frame.
    - The image could contain different background.
    - The image could contain different imposed occlusions.
- The dataset must cover painters' production as much as possible
- The system must recognize images in the range [128, … ,512] pixels.

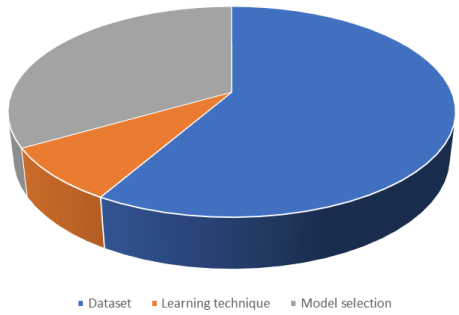
Non-functional requirements

- Build a generalized model (the model must be able to classify images of paintings not included in the training set)
- Build a robust model (the model must not be affected by frames, background, occlusion, etc.)
- Achieve the best performance on test set.

Even if these are not mandatory requirements for this project, it is still a good idea to try to create modular and easily maintainable software.

1. Project Base Architecture

In this chapter we want to discuss about the realization skeleton of the project, and so about the learning procedure adopted, the model made and the built dataset. For the model, we discuss only about the structure and not also about the chosen hyperparameters, which will be addressed in the next chapter.



Workload for each activity for the base architecture.

The dataset was completely built from scratch, collecting images of the three artists' paintings. For the learning procedure we decided to adopt the transfer learning in order to use some of the weights of an already trained network, only adding a head composed of some trainable layers. As base model we used one of the Keras model: the DenseNet121.

DataSet

A dataset in machine learning is, quite simply, a collection of data pieces that can be treated by a computer as a single unit for analytic and prediction purposes. This means that the data collected should be made uniform and understandable for a machine that doesn't see data the same way as humans do. For this, after collecting the data, it's important to preprocess it by cleaning and completing it, as well as annotate the data by adding meaningful tags readable by a computer. Moreover, a good dataset should correspond to certain quality and quantity standards.

The painters on whom the neural network was to carry out the recognition had to be Caravaggio, Manet and Van Gogh. So, during the creation of the dataset, we cover all the painters' productions. The choice of the images to be included in the dataset was made in order to satisfy the requirements on the type of images to be recognized. So, we collected images of pictures representing: the same "object" category (for example person…) but with a different style, different poses, different genres (religious subject, still life, landscape, etc.…). With regard to this aspect there were no problems as all three artists, even if in different numbers, had produced different paintings for each of these categories.

Another important requirement is that the images to be recognized may include the frame, the background or they can be acquired by different angles, or they can have some over imposed occlusions, for example text.

The rationale to collect the Dataset

Starting from these requirements, the entire dataset was formed from scratch following a well-defined process. First of all, we simply collected the paintings made by each of the artists, trying to cover the entire collection of paintings made by them, and also trying to cover all the categories mentioned above (person in different poses, religious subject, still life, landscape, etc.…).

The number of paintings created by each of the artists was quite unbalanced, but in this stage, we have not dealt with it. A solution to this problem was instead found during the learning procedure.

Since the goal was not the recognition of the subjects of the painting, but of the artists' style, we consider essential to include another type of images. In particular, we collect some images not representing the entire picture but only a part of this, for example some details. This was fundamental precisely because the artist's style was better visible in these details. Also, in this way we are sure that the choice for a specific artist was dictated by having recognized the style and not the painting.

Here are some examples:



In this phase we have taken care to bring all the images to the same size (512x512) and to make our dataset robust to any distortions, so that the trained model is not affected by frames, background, occlusions and has the capacity to generalize. To achieve this goal, we have followed two parallel paths. The first was to search some other images representing not only the artists' picture but the picture with frames, the painting photographed from different angles, foreseeing occlusions, such as people. So, the classic photos taken by visitors inside museums or during exhibitions. The second path is to use some scripts performing a static preprocessing to put frames, obstructions and make the size of the images the same.

Static preprocessing

As stated above we use some scripts for static preprocessing for different reason. First, we used it to convert the image in the RGB mode and to make the images of the same size (512x512). For resizing we have chosen not to crop the images but to add padding. In the addition of the padding there were several alternatives, we could have put a padding with monochrome background, but we did not choose this option because it could affect the learning, deviating the peculiar characteristics of the style of each artist. So, we have chosen to add padding in the "reflect" mode. This mode pads with the reflection of the vector mirrored on the first and last values of the vector along each axis. This choice was made because the aim of the project is precisely the recognition of the artists' style and not of the subjects depicted. So the addition of a padding of this type is more suited for our needs as it modifies the representation of the painting, which was not of our interest, but filled the empty areas with pixels in line with the artists' style without creating distortions.

Below there is an example representation, where it's possible to see the starting image with its own size and the resulting image after the process of adding the padding above descripted.

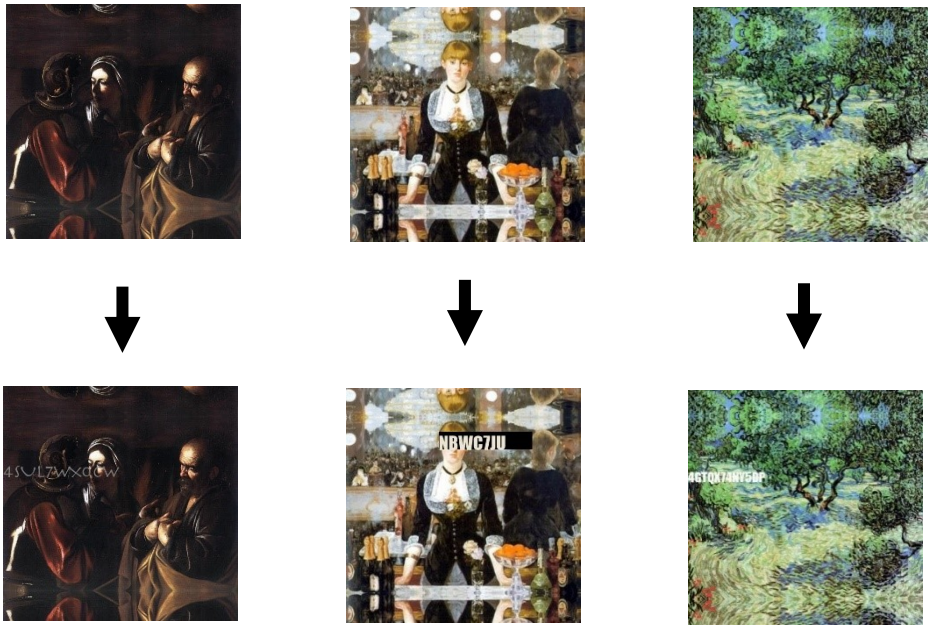




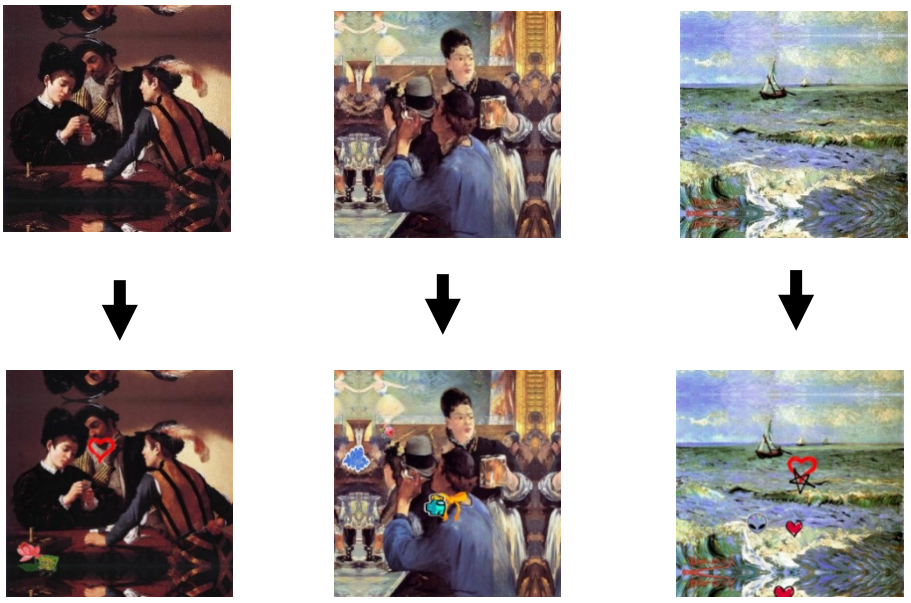

The second reason for which we have used this static preprocessing is to make the dataset robust against distortions. We prepared python scripts so that, for different percentages of basic collected images, we have applied other effects. One of the applied effects on the images containing only the paintings is the application of a frame. So, we collected different types of frames and then we applied that on about the 35% of the collected images on which we have performed the resize already descripted. Here are some examples:

Also, the static preprocessing was used to make the system resistant to various types of occlusions like text, or scribbles. So, we created python scripts that added these obstructions randomly on the 50% of the images containing only the picture.

Here are some examples of the text added:



Here there are some examples of the scribbles added:



So, the frames and the occlusions were applied only on the basic resized images. All the other images, representing shots in museums or in exhibitions have not been subjected to such actions because they are challenging enough.

Here there are some examples:

Through this preprocessing we are able to increase the number of elements in the dataset, that at the beginning was quite small, so because of this reason this preprocessing has been made in a static way. In fact, we also leave the basic version of images on which we applied modification in the dataset.

Have enough elements in the dataset to face up the problem was necessary in order to train the network and achieve good results.

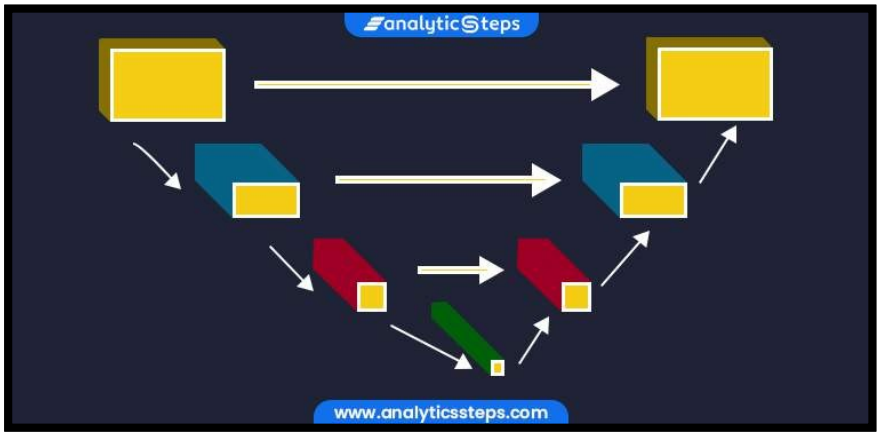
As described below other types of preprocessing are performed on the images, but at runtime throw data augmentation.

Learning Techniques

In this section there are all the techniques that were put into practice in the implementation phase; a description of each of these is given below.

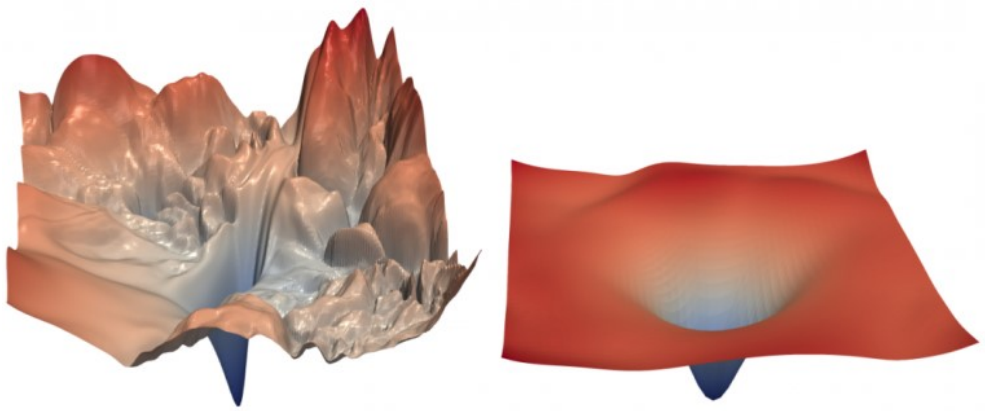
Skip connections

As the name suggests, the skip connections in deep architecture bypass some of the neural network layers and feed the output of one layer as the input to the following levels. It is a standard module and provides an alternative path for the gradient with backpropagation.



Skip connections using concatenation:

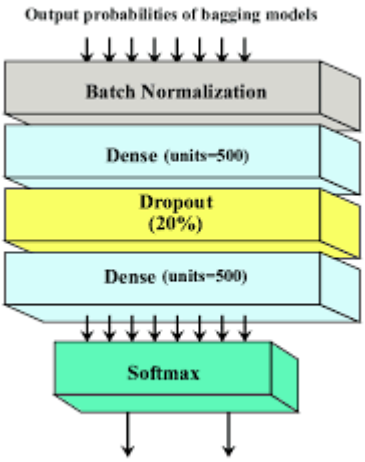
Low-level information is exchanged between the input and output, and it would be preferable to transfer this information directly across the network. Concatenation of prior feature maps is another method for achieving skip connections.



The loss surfaces

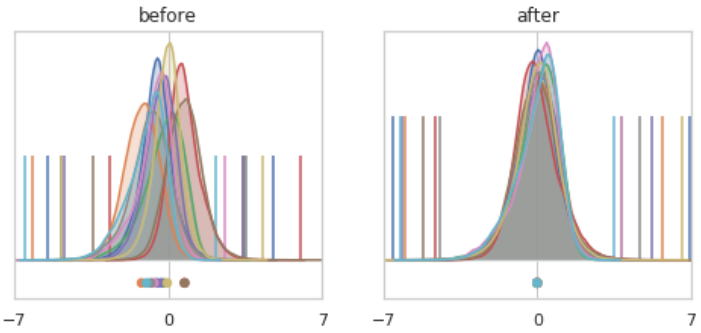
Batch normalization

This is a method used to make training of artificial neural networks faster and more stable through normalization of the layers' inputs by re-centering and re-scaling.



An effective technique for speeding up the learning of deep networks is to reduce the "coupling" between the layers by making the later layers independent of the mean and of the variance of the outputs of the earlier layers. In this way, the later layers are less affected by the fact that earlier layers are undergoing large weights updates (that modify the mean and variance of their output) during the initial phases of the training. Recently, some scholars have argued that batch normalization does not reduce internal covariate shift, but rather smooths the objective function, which in turn improves the performance.

However, at initialization, batch normalization in fact induces severe gradient explosion in deep networks, which is only alleviated by skip connections in residual networks. Others sustain that batch normalization achieves length-direction decoupling, and thereby accelerates neural networks.



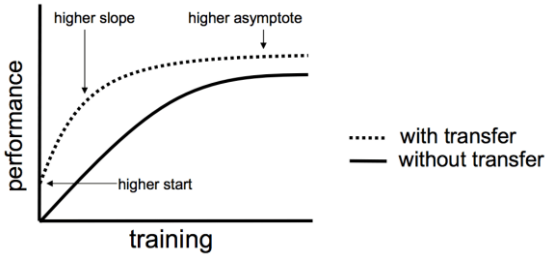
Transfer learning

Transfer learning is a machine learning technique where a model trained on one task is re-purposed on a second related task. This allows rapid progress or improved performance when modelling the second task, it will tend to work if the features are general (meaning suitable to both base and target tasks) instead of specific to the base task.

On some problems where you may not have many data, transfer learning can enable you to develop skillful models that you simply could not develop in the absence of transfer learning.

In fact, in our case the task that we had to manage was not so basically rich of images, so a model who had optimally trained layers on a similar task allowed us to reach:

1. **Higher start**. The initial performance (before refining the model) on the source model is higher than it otherwise would be.
2. **Higher slope.** The rate of improvement of performance during training of the source model is steeper than it otherwise would be.



3. **Higher asymptote.** The performance convergence of the trained model is better than it otherwise would be.

The choice of source data or source model is an open problem and may require domain expertise and/or intuition developed via experience.

Pre-trained Model Approach

To use this technique, we have to:

1. **Select Source Model.** A pre-trained source model is chosen from available models. Many research institutions release models on large and challenging datasets that may be included in the pool of candidate models from which to choose.
2. **Reuse Model.** The pre-trained model can then be used as the starting point for a model on the second task of interest. This may involve using all or parts of the base model, depending on the modeling technique used.
3. **Tune Model.** The model may need to be adapted or refined on the input-output pair data available for the task of interest.

This type of transfer learning is common in the field of deep learning.

Selected Model

Our project is based on CNNs (Convolutional Neural Networks), as their use in image classification activities is very prolific. So, let's see briefly the architecture of a CNN and how it works. A CNN mainly consists of a number of convolutional layers, pooling layers and fully connected layers. A complete CNN architecture can be formed by composing layers in a specific order. Let's see the function of the different layers:

Convolutional layer: Convolutional layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input. Neurons that belong to the same layer share the same weights. Weight sharing increases learning efficiency by greatly reducing the number of free parameters being learnt. The constraints on the model enable CNN to achieve better generalization on vision problems.

Pooling layer: The function of pooling layer is to progressively reduce the spatial size of the representation to reduce the number of parameters and computation in the network. Max pooling and average pooling are typical types. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2.

Fully connected layer: The output from the convolutional and pooling layers represents high-level features of the input images. The purpose of the fully connected layer is to use these features for classifying the input image into various classes based on the training dataset.

Neural networks have to implement complex mapping functions hence they need activation functions that are non-linear in order to bring in the much-needed non-linearity property that enables them to approximate any function. Activation function is applied after convolutional layer and fully-connected layer. The most commonly used types of activation functions are Sigmoid, tanh, and ReLU function.

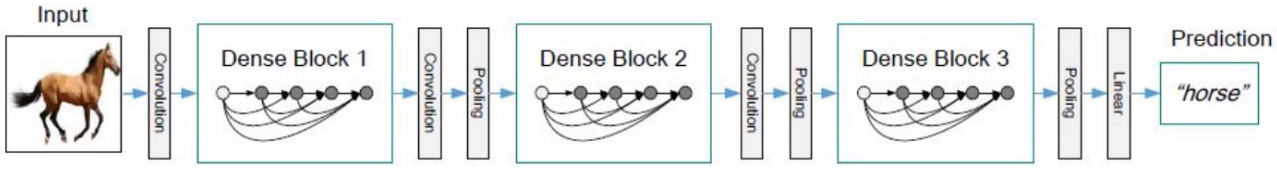
The training process contains two steps: forward propagation and backward propagation. The forward propagation means that training images goes through a series of convolution, activation function, pooling operations and fully-connected layer to output probabilities for each class. After calculating the total error between target probability and output probability, backpropagation is used to calculate the gradients of the error with respect to all weights in the CNN and use gradient descent to update the trainable parameters to minimize the output error.

Our model is based on transfer learning. The network on which we applied transfer learning is DenseNet121, a model of the DenseNet group of models designed to perform image classification. Let's introduce the use of DenseNets and the problems that they resolve. In a traditional feed-forward Convolutional Neural Network (CNN), each convolutional layer except the first one (which takes in the input), receives the output of the previous convolutional layer and produces an output feature map that is then passed on to the next convolutional layer. Therefore, for 'L' layers, there are 'L' direct connections; one between each layer and the next layer. However, as the number of layers in the CNN increase (as they got deeper) the 'vanishing gradient' problem arises. This means that as the path for information from the input to the output layers increases, it can cause certain information to 'vanish' or get lost which reduces the ability of the network to train effectively. DenseNets resolve this problem by modifying the standard CNN architecture and simplifying the connectivity pattern between layers. In a DenseNet architecture, each layer is connected directly with every other layer, hence the name Densely Connected Convolutional Network. For 'L' layers, there are L(L+1)/2 direct connections. Components of DenseNet include:

- Connectivity
- DenseBlocks
- Growth Rate
- Bottleneck layers

Connectivity: In each layer, the feature maps of all the previous layers are not summed but concatenated and used as inputs. Consequently, DenseNets require fewer parameters than an equivalent traditional CNN, and this allows for feature reuse as redundant feature maps are discarded. So, the l-th layer receives the feature-maps of all preceding layers, x0,...,xl-1, as input, where [x0,x1,...,xl-1] is the concatenation of the feature-maps, i.e. the output produced in all the layers preceding l (0,...,l-1).

Dense blocks: the use of the concatenation operation is not feasible when the size of feature maps changes. However, an essential part of CNNs is the down-sampling of layers which reduces the size of feature-maps through dimensionality reduction to gain higher computation speeds. To enable this, DenseNets are divided into DenseBlocks, where the dimensions of the feature maps remain constant within a block, but the number of filters between them is changed. The layers between the blocks are called Transition Layers which reduce the number of channels to half of that of the existing channels. For each layer Hl is defined as a composite function which applies three consecutive operations: batch normalization (BN), a rectified linear unit (ReLU) and a convolution (Conv).



In the above image, a deep DenseNet with three dense blocks is shown. The layers between two adjacent blocks are the transition layers which perform down-sampling (i.e. change the size of the feature-maps) via convolution and pooling operations, whilst within the dense block the size of the feature maps is the same to enable feature concatenation.

Growth rate: one can think of the features as a global state of the network. The size of the feature map grows after a pass through each dense layer with each layer adding 'K' features on top of the global state (existing features). This parameter 'K' is referred to as the growth rate of the network, which regulates the amount of information added in each layer of the network. If each function H-l produces k feature maps, then the lth layer has:

$$k_l = k_0 + k * (l - 1)$$

input feature-maps, where k0 is the number of channels in the input layer. Unlike existing network architectures, DenseNets can have very narrow layers.

Bottleneck layers: Although each layer only produces k output feature-maps, the number of inputs can be quite high, especially for further layers. Thus, a 1x1 convolution layer can be introduced as a bottleneck layer before each 3x3 convolution to improve the efficiency and speed of computations.

These are the various architectures of the DenseNets implemented for the ImageNet database:

Layers	Output Size	DenseNet-121	DenseNet-169	DenseNet-201	DenseNet-264
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	[1 × 1 conv, 3 × 3 conv] × 24	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48	[1 × 1 conv, 3 × 3 conv] × 64
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	[1 × 1 conv, 3 × 3 conv] × 16	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			

DenseNet-121 has the following layers:

- 1 7x7 Convolution
- 58 3x3 Convolution
- 61 1x1 Convolution
- 4 AvgPool (3 2x2 and 1 7x7)
- 1 Fully Connected Layer

In short, DenseNet-121 has 120 Convolutions and 4 AvgPool.

Our model has been built using pre-trained DenseNet121 and connecting some final layers after that; only the first 10 layers of the pre-trained network are frozen (we don't change their weights); we added consecutively:

- 1 Dense Layer
- 1 2x2 Average Pooling Layer
- 1 2x2 Max Pooling Layer
- 1 Dropout Layer
- 1 Dense Layer

Dense layer is the regular deeply connected neural network layer. It is most common and frequently used layer. Dense layer does the below operation on the input and return the output.

output = activation(dot(input, kernel) + bias)

where:

- **input** represents the input data
- **kernel** represents the weight data
- **dot** represents numpy dot product of all input and its corresponding weights
- **bias** represents a biased value used in machine learning to optimize the model
- **activation** represents the activation function.

Our dense layer has a dimensionality of the output space equal to 128 and ReLU activation function.

Our **pooling layers** are applied with filters of size 2x2 without strides.

Dropout is a technique used to prevent a model from overfitting. Dropout works by randomly setting the outgoing edges of hidden units (neurons that make up hidden layers) to 0 at each update of the training phase. Our dropout layer has 0.3 value that is the fraction of the input units to drop.

The final layer is dense layer with 3 units (i.e. the number of classes for our project) and Softmax activation function.

2. Learning Procedure

In this chapter we will discuss the learning procedure adopted, so the training phase and the chosen hyperparameters. During this phase it is necessary to find a solution to different encountered problems. First of all, it was necessary to be certain that the network will work also in difficult situations, so that it is resistant to different forms of distortions applied on images, like rotations, different angles or different levels of brightness. It was also important to mitigate the fact that the number of images in the dataset is not very large and it could not be increased since the production of the artists had almost been fully covered. A solution to these two problems has been proposed through the use of a dynamic Data Augmentation.

Also, there is another important problem due to the imbalance of the dataset between the picture of the artists. This has been solved by the correct choice of some hyperparameters.

Preprocessing Pipeline (Dynamic Augmentation)

A problem with our dataset was that we had not much data, which is justified because the number of pictures made by the artists was limited, and in this case, the Image Augmentation technique is a great way to resolve this problem. Through Image Augmentation the network becomes also able to recognize the pictures in different contexts and with different positions.

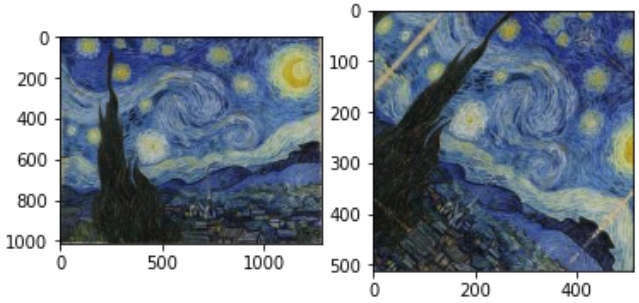
Image augmentation applies different transformations to original images, resulting in multiple transformed copies of the same images. Each copy, however, is different from the others in certain aspects depending on the applied augmentation techniques like shifting, rotating, flipping, etc.

This technique permits to obtain new transformed images from the dataset through Keras's ImageDataGenerator class in on the fly while the model is still training. This class can apply any random transformation on each training image as it is passed to the model. This not only makes the model robust but will also save up on the overhead memory (because all Dataset isn't loaded into the memory, but every step the Generator returns a predefined number of randomly modified images).

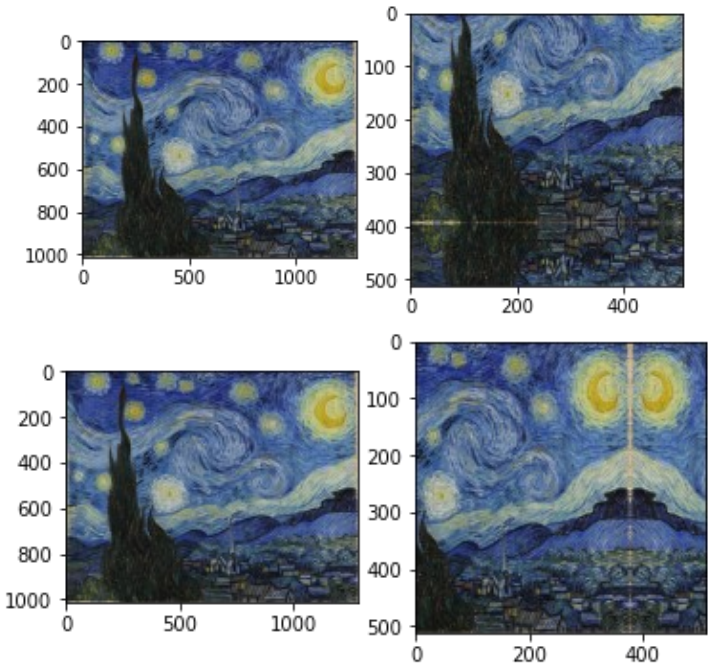
This Keras class provides a host of different augmentation techniques like standardization, rotation, shift, flips, brightness change, and many more.

Now we see which augmentation we applied to our Dataset and why:

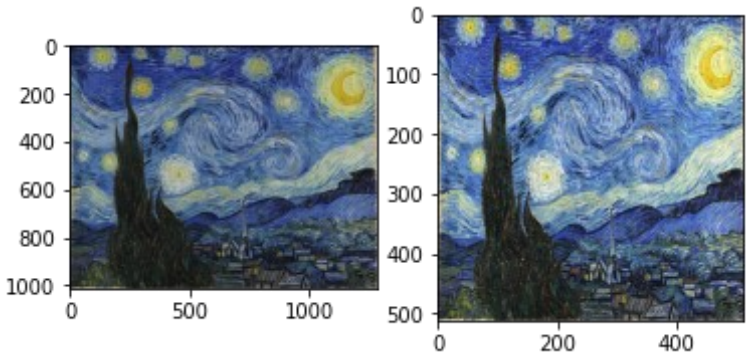
- Random Rotation
    - Image rotation is one of the widely used augmentation techniques and allows the model to become invariant to the orientation of the object; this allows for randomly rotating images through any degree between 0 and 360 depending on an integer value specified as rotation_range argument. When the image is rotated, some pixels will move outside the image and leave an empty area that needs to be filled in, we will talk about the "fill_mode" after. In our case we used a value of 45 because a higher value would not have been real.



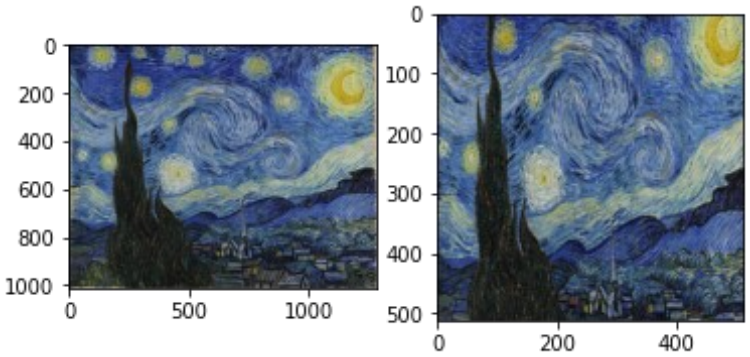
- Random Shift
    - It may happen that the object may not always be in the center of the image. To overcome this problem, we can shift the pixels of the image either horizontally or vertically; this is done by adding a certain constant value to all the pixels, in our case this area will be filled in relation to the “fill_mode”. ImageDataGenerator class argues height_shift_range for a vertical shift of image and width_shift_range for a horizontal shift of image. The value is a float number, that indicates the percentage of width or height of the image to shift. In our case we used a value of 0.3 for the width shift and 0.3 for the height shift.



- Random Brightness
    - It randomly changes the brightness of the image. It is also a very useful augmentation technique because most of the time our object will not be under perfect lighting conditions. So, it becomes imperative to train our model on images under different lighting conditions. Brightness can be controlled in the ImageDataGenerator class through the brightness_range argument. It accepts a list of two float values and picks a brightness shift value from that range. Values less than 1.0 darkens the image, whereas values above 1.0 brighten the image. In our case we used a value range of 0.5, 1.5 for the brightness.

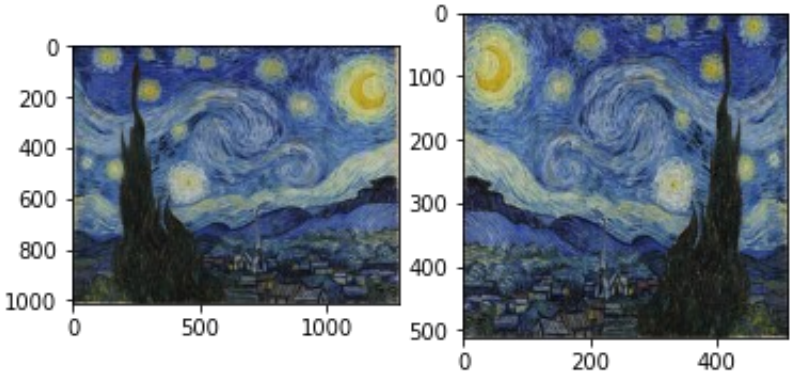


- Random Zoom
    - The zoom augmentation either randomly zooms in on the image or zooms out of the image. ImageDataGenerator class takes in a float value for zooming in the zoom_range argument. We provided a list with two values specifying the lower and the upper limit for this parameter; in our case those values specify only a zoom-in because zoom-out gives problem for recognizability of images (values minor than 1 are zoom-in, greater than 1 zoom-out). In our case we used a value range of 0.7, 1 for the zoom.



- Fill Mode
    - How we said previously, some transformations move the original images and some space of the image remains empty, so with this parameter we choose how to manage this situation. In the specific, we choose to fill the space with the “reflect” mode. This mode uses the opposite pixels of the image to fill the space. This method was used also to create the images presented in this chapter.

- Random Flip
    - Flipping images is also a great augmentation technique and it makes sense to use it with a lot of different objects**. ImageDataGenerator** class has the parameters **horizontal_flip** and **vertical_flip** for flipping along the vertical or the horizontal axis. However, this technique should be according to the object in the image. For example, vertical flipping of a picture would not be a sensible thing compared to doing it for a symmetrical object like a ball or something else, so we use only **horizontal_flip.**



The **flow_from_directory()** method allows us to read the images directly from the directory and augment them while the neural network model is learning on the training data. To use this method, we need to provide the directory, target size, batch size, and class mode. The first is the directory where the data are stored, the second is the size of the image, the third is the batch size that indicates how many elements to return for each call, and the class mode indicates which type of problem we are managing, so if is binary problem or multi-class problem.

After all that we have the generator for the training set with augmentation that will be passed to the **model.fit_generator**.

Training Hyperparameters

Hyperparameters are defined as the parameters explicitly defined by the user to control the learning process. These hyperparameters are used to improve the learning of the model, and their values are set before starting the learning process of the model.

Split between training and validation

The Training Set

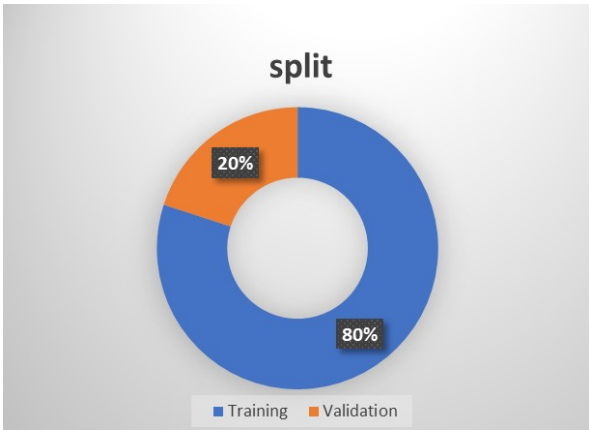
Is a set of examples to each of which is associated an answer, the value of an attribute-goal, i.e. a categorical value, i.e. a class, or a numeric value. Such examples are used to train the supervised predictive model to determine the target value for new examples. On our base Training set are applied lots of dynamic transformation in order to achieve best performance in compliance with the functional requirements.

The Validation Set

The validation set is a set of data, separate from the training set, that is used to validate our model performance during training. This validation process gives information that helps us tune the model's hyperparameters and configurations accordingly. It is like a critic telling us whether the training is moving in the right direction or not. The model is trained on the training set, and, simultaneously, the model evaluation is performed on the validation set after every epoch.

The main idea of splitting the dataset into a validation set is to prevent our model from overfitting i.e., the model becomes really good at classifying the samples in the training set but cannot generalize and make accurate classifications on the data it has not seen before.

At the end of careful analysis, we decided to apply the following division:



Batch size and number of epochs

Batch size represents the number of training samples utilized in one iteration.

In general, we do not have a precise guideline on how to set this parameter, usually larger batch sizes result in faster progress in training, but don't always converge as fast. Smaller batch sizes train slower but can converge faster. It's definitely problem dependent; it also relates to the number of epochs set for training.

This parameter will also have to be tested in regard to how our machine is performing in terms of its resource utilization when using different batch sizes.

At the end of a careful analysis, we decided to apply to the parameter **batch_size a value equal to 8**.

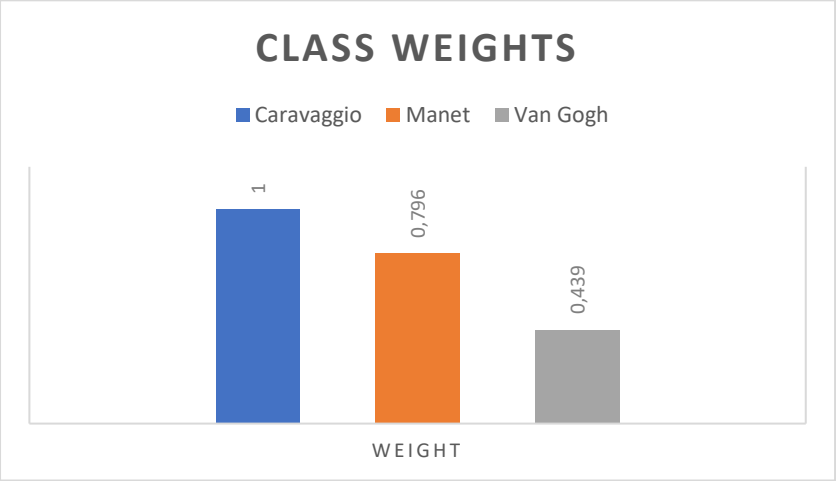
Weights associated to artists' error

Different class weights are used when you have an imbalanced dataset and want to improve single-label classification results, as in our case.

During model training, a total loss is computed for each batch, and the model parameters are then iteratively updated in a direction that reduces this loss. By default, each sample counts equally into this total loss. With defined class weights, the sum is replaced instead by a weighted sum so that each sample contributes to the loss proportionally to the sample's class weight.

For this analysis, first of all we have imposed for each artist a value that is inversed proportional with number of samples that we had for him. After we had normalized these values and finally add a last parameter associated to the difficult with the recognizability of the artists' style.

At the end of careful analysis, we decided to apply:



Parameter in dropout layers

The Dropout layer randomly sets input units to 0 with a frequency of rate at each step during training time, which helps prevent overfitting. Inputs not set to 0 are scaled up by 1/(1 - rate) such that the sum over all inputs is unchanged. The Dropout layer only applies when training is set to True such that no values are dropped during inference.

In our case we had to set only one parameter of 1 dropout layer, and at the end of careful analysis we decided to apply: **rate = 0.3**.

Dimension of dense layers

Dense layer is the regular deeply connected neural network layer. It is most common and frequently used layer. The *units* arguments is the first parameter of the dense layer and represent the number of units, it affects the output layer.

In our case we had to set only one parameter for the first dense layer attached to the output of the base model, and at the end of careful analysis we decided to apply: **units = 128.**

Choose of trainable layers

In Keras, each layer has a parameter called "trainable". For freezing the weights of a particular layer, we should set this parameter to False, indicating that this layer should not be trained.

For the base model we used we sets that only the first **10 layers are "untrainable"** leave to the other the possibilities to change during training phase.

Loss and activation function

The loss function is the function that computes the distance between the current output of the algorithm and the expected output. It's a method to evaluate how your algorithm models the data. It can be categorized into two groups. One for classification (discrete values, 0,1,2…) and the other for regression (continuous values). As loss function, we have chosen to use the categorical cross-entropy. The cross-entropy as the Log Loss function (not the same but they measure the same thing) computes the difference between two probability distribution functions. The cross-entropy is a class of Loss function most used in machine learning because that leads to better generalization models and faster training. Cross-entropy can be used with binary and multiclass classification problems (as in this project). There are different types of cross-entropy:

- Binary cross-entropy: for binary classification problem
- Categorical cross-entropy: binary and multiclass problem, the label needs to be encoded as categorical, one-hot encoding representation (for 3 classes: [0, 1, 0], [1,0,0], …)
- Sparse cross-entropy: binary and multiclass problem (the label is an integer — 0 or 1 or … n, depends on the number of labels)

Range of values for this class of Loss function:

- **0.00**: Perfect probabilities
- **< 0.02**: Great probabilities
- **< 0.05**: In a good way
- **< 0.20**: Great
- **> 0.30**: Not great
- **1.00**: Hell
- **> 2.00** Something is not working

The categorical cross-entropy loss function calculates the loss of an example by computing the following sum:

$$\text{Loss} = -\sum_{i=1}^{\substack{\text{output}\\ \text{size}}} y_i \cdot \log \hat{y}_i$$

Where yi s the i-th scalar value in the model output, yi is the corresponding target value, and output size is the number of scalar values in the model output. This loss is a very good measure of how distinguishable two discrete probability distributions are from each other. In this context, yi is the probability that event ii occurs and the sum of all yi is 1, meaning that exactly one event may occur. The minus sign ensures that the loss gets smaller when the distributions get closer to each other. The categorical cross-entropy is well suited to classification tasks, since one example can be considered to belong to a specific category with probability 1, and to other categories with probability 0.

The Softmax function is a function that turns a vector of K real values into a vector of K real values that sum to 1. The input values can be positive, negative, zero, or greater than one, but the Softmax transforms them into values between 0 and 1, so that they can be interpreted as probabilities. If one of the inputs is small or negative, the Softmax turns it into a small probability, and if an input is large, then it turns it into a large probability, but it will always remain between 0 and 1.

The Softmax function is sometimes called the softargmax function, or multi-class logistic regression. This is because the Softmax is a generalization of logistic regression that can be used for multi-class classification,

and its formula is very similar to the sigmoid function which is used for logistic regression. The Softmax function can be used in a classifier only when the classes are mutually exclusive.

Many multi-layer neural networks end in a penultimate layer which outputs real-valued scores that are not conveniently scaled and which may be difficult to work with. Here the Softmax is very useful because it converts the scores to a normalized probability distribution, which can be displayed to a user or used as input to other systems. For this reason, it is usual to append a Softmax function as the final layer of the neural network as we have done in the project.

Softmax is the only activation function recommended to use with the categorical cross-entropy loss function.

Strictly speaking, the output of the model only needs to be positive so that the logarithm of every output value yi exists. However, the main appeal of this loss function is for comparing two probability distributions. The Softmax activation rescales the model output so that it has the right properties.

For the first layer after the DenseNet layers we used the ReLU activation function. The ReLU is the most used activation function in the world right now, since it is used in almost all the convolutional neural networks or deep learning. The rectified linear activation function or ReLU is a non-linear function or piecewise linear function that will output the input directly if it is positive, otherwise, it will output zero. ReLU is used in the hidden layers instead of Sigmoid or tanh as using Sigmoid or tanh in the hidden layers leads to the infamous problem of "Vanishing Gradient". The "Vanishing Gradient" prevents the earlier layers from learning important information when the network is backpropagating.

The sigmoid which is a logistic function is more preferrable to be used in regression or binary classification related problems and that too only in the output layer, as the output of a sigmoid function ranges from 0 to 1. Also, Sigmoid and tanh saturate and have lesser sensitivity.

Some of the advantages of ReLU are:

- **Simpler Computation**: Derivative remains constant i.e 1 for a positive input and thus reduces the time taken for the model to learn and in minimizing the errors.
- **Representational Sparsity**: It is capable of outputting a true zero value.
- **Linearity**: Linear activation functions are easier to optimize and allow for a smooth flow. So, it is best suited for supervised tasks on large sets of labelled data.

Some of the disadvantages of ReLU are:

- **Exploding Gradient**: This occurs when the gradient gets accumulated, this causes a large difference in the subsequent weight updates. This as a result causes instability when converging to the global minima and causes instability in the learning too.
- **Dying ReLU**: The problem of "dead neurons" occurs when the neuron gets stuck in the negative side and constantly outputs zero. Because gradient of 0 is also 0, it's unlikely for the neuron to ever recover. This happens when the learning rate is too high or negative bias is quite large.

3. Conclusions

Test-Set

After the training procedure, we tried to build a little test-set with new images from the three artists (obviously independent from the ones used in the training). The objective in this phase was to test our trained model on new images, also a bit harder to be guessed, to have a sort of feedback from the network on the real world. In general, a test-set is a data set that is independent of the training data set, but that follows the same probability distribution as the training data set. A test set is therefore a set of examples used only to assess the performance (i.e. generalization) of a fully specified classifier. To do this, the final model is used to predict classifications of examples in the test set. Those predictions are compared to the examples' true classifications to assess the model's accuracy. In a scenario where both validation and test datasets are used, the test data set is typically used to assess the final model that is selected during the validation process as we have done in this project.

Caravaggio's test set picture

Manet's test set picture

VanGogh's test set picture

Problems Faced

In the previous chapters we have faced several problems relative to the picture and to the objective of the project.

Problems relative to the picture and to the dataset:

- The quantity of pictures, in fact the artist production (especially in the case of Caravaggio's pictures) wasn't too expanse.
    - We manage this problem through two different Data Augmentation (1 static, 1 dynamic). The resulting dataset is thus much larger and this has allowed us to train the network more in depth, even with images that are different from the originals, in terms of: frames, position, angles, brightness etc...
    - Using Transfer Learning technique, we are able to mitigate the presence of few images, exploiting the weights of the already carefully trained model
- The different quantity of pictures produced by different artists
    - We manage this problem setting different weights for the classes representing the three artists, in fact, we utilize those weights in case the network didn't recognize a specific artist well, increasing the "cost" for the error made.

- The functional requirement that asks us to recognize images even in the presence of imposed occlusions

    - We manage this problem through the static Data Augmentation, in fact, we artificially entered text on the images in random positions, with random length, with random dimensions and font.
- The functional requirement that asks us to recognize images even with different frames
    - We manage this problem through the static Data Augmentation, in fact, we manually entered random typical frames on the images.
- The functional requirement that asks us to recognize images even containing artists' pictures in different positions and backgrounds
    - We manage this problem through research of images containing artists picture but immersed in street, museum or exhibitions context.
- We apply preprocessing to images to resize them all to 512x512, so we have faced the choice of what padding apply in case was necessary
    - We manage this problem applying paddings who reflect the content of the image to leave in the image always a part of picture containing the style of the artist.
- Recognition of pictures with different brightness, shift, rotation, zoom
    - This is a problem we considered while we had studied the possible dynamic Augmentation keras provides. So we chose parameters to include this type of Augmentation.
- Recognition of poorly defined images
    - This is a problem we encountered later, which we hadn't paid attention to before. We discover that if we give to the network a poor defined image the network didn't correctly recognize the author. However, this problem was not a prerogative of our project, in fact, the neural network must recognize the artist's style and with a poor defined image this result difficult even for a human being. We decided to not correct this problem because we thought that poorly defined images alter the artists' styles and so it would make no sense to submit them to the network.

Problems relative to the model:

- The choose of base model for Transfer Learning
    - There are lots of model in literature adapt to this task, so it was really difficult to choose the best one for our case, only trying different models, among those that seemed most appropriate to us, on the basis of the purposes for which they were created and the use that had been made of them in the various projects, we were able to choose.
- The choose of layers at the head of our base model
    - There are lots of possible configuration to perform a ternary classification, so it was really difficult to choose the best one for our case, only trying many of them we have choose one.

Problems relative to the hyperparameters:

- The choose of batch size for our learning procedure
    - We have chosen the best couple between batch size and number of epochs in order to achieve best performance in minor time.
- The choose of weights to give to different artists error
    - We have chosen the best values best representing as the relationship between the number of frames as the difficult associated with the recognizability of the artist's hand in order to achieve best performance.
- The choose of parameter in dropouts layers added at the head of model
    - We have chosen the best values in order to achieve best performance with a not so long training but avoiding overfitting.
- The choose of dimension of dense layers added at the head of model
    - We have chosen the best values in order to achieve best performance in minor time.
- The choose of trainable layers in the base model
    - We have chosen the best numbers of trainable layers, respecting the number of images in the dataset and therefore the possibility of conducting a more or less invasive training; in order to achieve best performance in minor time.

Final Considerations

To develop a good model for the assigned task we analyzed various networks and tested a lot of different hyperparameters. Despite many networks have good result in image classification, in our project this has not always been a guarantee of success. From the results obtained, it is possible to observe the difficulties and advantages of using already trained convolutional neural network architectures, through transfer learning, outside of their original sample space.

The learning technique we used is advantageous when the time for development is taken into account, since all the architectures used were exceptional in their specific tasks and have already been extensively studied. Furthermore, this approach is so useful when the available dataset is not sufficiently big to train and to develop a good model because the base model used is already well-trained, so it only has to adapts its weights for the specific task. However, it was also seen that this is not a guarantee that these already trained networks will perform well in any task.

Another cause of difficulty in choosing the network is due to the fact that image classification is a specific area of recognition problems, each application that falls into said area has its set of difficulties and characteristics that, also some well-established state-of-the-art architectures, may not excel in. During the tests all networks were submitted to the same training conditions in order to compare them in solving the proposed task, without favoring any specific network. However, when thoroughly analyzing each network architecture and their scope, it was clear that DenseNet121 architecture had the best results in terms of accuracy. Other architectures like ResNet, VGG and EfficientNet were tested and also obtained good results, but the one we have chosen was the best not only for accuracy values but also for the result on our created final Test-Set.

Finally, this project explored the adaptability of prominent convolutional neural networks architectures present in the literature when applied to a different image classification problem respect to their original one. These types of explorations demonstrate how intense and variegated are the application scopes of this systems and how full of evolution and new knowledge they are.

Final result:

In neural networks, the loss function quantifies the difference between the expected outcome and the outcome produced by the model. Then naturally, the main objective in a learning model is to reduce (minimize) the loss function's value with respect to the model's parameters by changing the weight vector values through different methods, such as backpropagation algorithm. In fact, backpropagation works by calculating the gradient of the loss function, which points us in the direction of the value that minimizes the loss function. It relies on the chain rule of calculus to calculate the gradient backward through the layers of a neural network. Using gradient descent, we can iteratively move closer to the minimum value by taking small steps in the direction given by the gradient.

Loss value implies how well or poorly a certain model behaves after each iteration of the training. Ideally, one would expect the reduction of loss after each, or several, iteration(s).

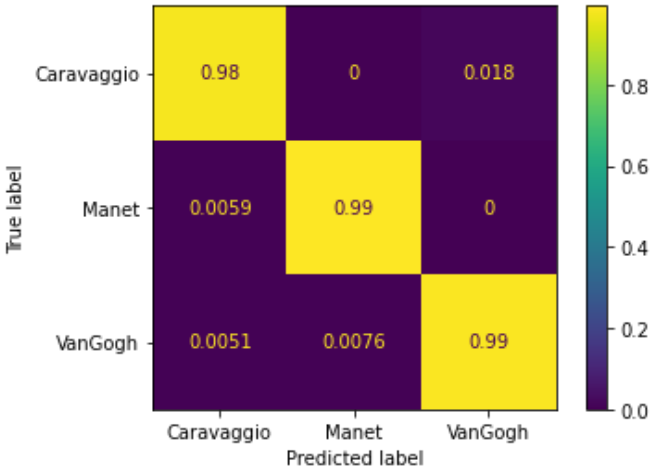
The accuracy is used to measure how good a system is. The accuracy of a model is usually determined after the model parameters are learned and fixed and no learning is taking place. Then the test samples are fed to the model and the number of mistakes (zero-one loss) the model makes are recorded, after comparison to the true targets. Then the percentage of misclassification is calculated.

Below it is possible to see the obtained value of loss and of accuracy on the validation set. The value of accuracy is really very high, but it is necessary to consider that the evaluation has been made on the validation set. Obviously, the data used to evaluate our system performance were challenging enough in order to get as close as possible this value to that obtained in the real word use and so in the test set.

```
loss: 0.040169
accuracy: 0.988095
```

Confusion Matrix:

The confusion matrix is a specific table layout that allows visualization of the performance of a network. Each row of the matrix represents the instances in an actual class while each column represents the instances in a predicted class, or vice versa. It is useful to understand the eventually problem of the network in some specific recognition. In our case, for example, the problem could have been an excessive polarization on some mistakes made on the same artists. As it is possible to see in the matrix shown below, there is no polarization on errors, so there aren't artists more exposed to misclassifications.



We have also tested this aspect by analyzing the error made by the neural network on the validation set. The noticed that the misclassification were enough equally distributed, probably some more error have been made making confusion between Van Gogh and Manet. This kind of mistake was something we expect because there are some paintings of Van Gogh whose style is very confusing with Manet's style, with little sharp and little marked edges. Caravaggio's paintings, on the other hand, have very clearly distinguishable features with clear and precise lines.

Epoch accuracy and loss:

Throw Tensorboard and logs files it is possible to visualize in a graphic way the adaptation of the network during training phase in terms of loss and accuracy in each epoch.

The main problem when defining the model is the risk of overfitting or underfitting. So, respectively, that the model is too complex so has no capacity of generalization or too simple and so is not sufficient. It's always necessary to find a trade-off. As it is possible to see in the graphs of error and accuracy shown above, we did not have problem with overfitting. Starting from this type of graphs it is possible to check the overfitting; we will have it if, as the epochs increase, the accuracy values on the training set continue to increase while those on the validation set do not and therefore a large gap is created between the two lines. The same considerations are also valid for the loss parameters.

Downstream of these assessments, we can affirm that all work has been completed according to the project plan and scope.