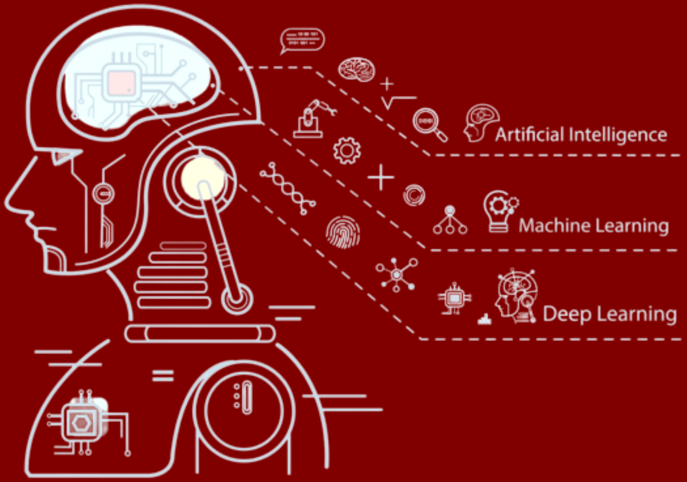
Università degli Studi di Salerno
DIEM, a.a. 2021-22
Course: Machine Learning
Lecturers: Pasquale Foggia, Diego Gragnaniello



PROJECT WORK ON *PAINTER STYLE RECOGNITION* BY *GROUP 29*

Salvatore Grimaldi	0622701742	s.grimaldi29@studenti.unisa.it
Enrico Maria Di Mauro	0622701706	e.dimauro5@studenti.unisa.it
Allegra Cuzzocrea	0622701707	a.cuzzocrea2@studenti.unisa.it
Andrea De Gruttola	0622701880	a.degruttola@studenti.unisa.it

Contents

Introduction

The main purpose of this report is describing the steps followed in order to obtain a **classifier** able to recognize *photos of paints of three different artists*:

- Caravaggio
- Manet
- Van Gogh

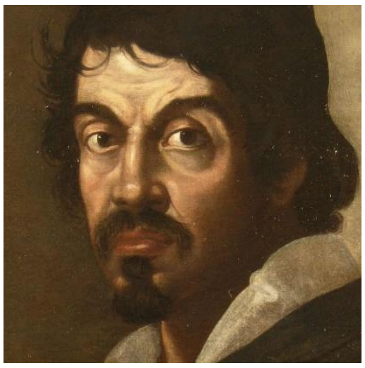


Figure 1. Caravaggio (1571-1610)

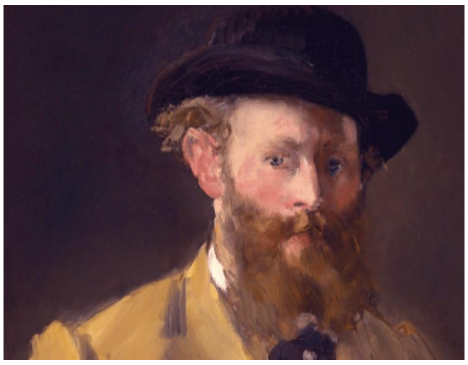


Figure 2. Manet (1832-1883)

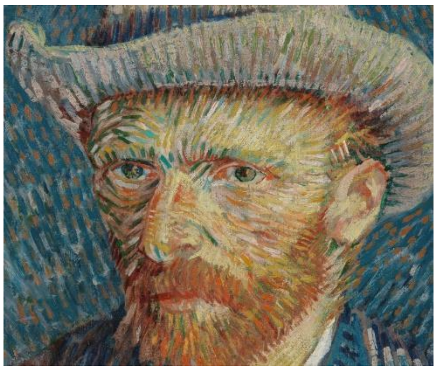


Figure 3. Van Gogh (1853-1890)

In other words, our purpose is to carry out the task of **artist identification**, which consists of identifying the artist of a painting given no other information about it. Nowadays it is an important requirement for cataloguing art, especially as paintings are increasingly digitized. Artist identification in general is not an easy task to perform by a neural network because identifying an artist does not just require object or face detection but a more complex style recognition based on colors, brush strokes, subjects, etc.

In the following pages the problem faced is going to be described in a detailed way, with particular attention to:

- The rationale to collect the *dataset*
- How the dataset *split* into training and validation sets was done
- The *pre-processing* pipeline adopted
- The network *architecture* selected
- The *training hyperparameter* setting, including the loss function
- Performance *analysis* through quantitative indicators

All the design choices and the obtained results are going to be analyzed and explained in the light of the theoretical and practical knowledge acquired during the **Machine Learning** course held by prof. Pasquale Foggia and prof. Diego Gragnaniello at Diem, University of Salerno.

Setup

For all our implementations and experiments we used **Colaboratory** (or "Colab" for short), a product from Google Research that allows anybody to write and execute arbitrary Python code through the browser and is especially suited to machine learning thanks to the fact that it provides a good amount of computational resources in terms of ram and gpu. In particular, we used a 12GB ram, a 78GB disk and the most of the times the assigned gpu was a Nvidia Tesla-K80. All the tests were performed loading the dataset from a shared **Google Drive** folder. Moreover, we used **Keras**, that is an open-source software library that provides a Python interface for artificial neural networks. Keras acts as an interface for the TensorFlow library.

Dataset building

The first fundamental step is **dataset building**. The purpose is covering the artistic productions of the involved painters as much as possible, considering images containing frames, backgrounds, watermarks, texts, occlusions and characterized by different angles. Moreover, photos may represent different genres, such as religious subjects, still life, landscapes, etc., and people in different poses.

To reach this target we decided to mix the results given by:

- ***"Handmade" search.*** Each team member focused on a painter. In particular, given that the most prolific artist among the considered ones is Vincent Van Gogh, two group members were assigned to search his works online, while the other two group members focused individually on the other artists, Caravaggio and Manet. Handmade search gave us the opportunity to select carefully the *most qualitative* photos of the most famous paintings, at the same time paying attention to find images characterized by watermarks, texts, backgrounds and several angles. Our purpose was collecting a *pretty balanced* dataset *in terms of these characteristics* for each artist, so that generalization would have been easier to reach and not affected by the fact that for a particular painter the percentage of photos with a particular characteristic (watermarks, backgrounds, frames, etc.) was much bigger or smaller than the percentage of photos with the same characteristic of the other two artists.
  Good sources of information and photos were: wikipedia.it, caravaggio-foundation.org, manetedouard.org, this archive of Van Gogh's paintings, and, of course, google images
- ***Search realized by a smart Google Images crawler.*** Since we wanted to *cover the productions of each painter as much as possible*, it was clear that only handmade search of qualitative pics was not enough. This is why we decided to use a *smart crawler*[1] to get more images from web. The samples got from this kind of search were even more *variegate* thanks to the chance of specifying image dimensions, presence of frames, texts, etc.

After that we made a **careful selection** among all the images found, in order to delete:

- Images *too small* (e.g.: under the estimated resolution of test set images)
- Images whose quality was *very low*. In fact, *resolution* of training samples is fundamental for an image classifier
- *Copies* of exactly the same images
- *Wrong images* because representing works not painted by Caravaggio, Manet or Van Gogh

The aim was obtaining only *useful* photos so that the training phase would not have been problematic, and generalization could be reached.

Given that the samples found online characterized by over imposed texts were not so many and variegate, we decided to create new *custom samples* with these characteristics. This is the reason why we used a very easy and smart online tool, whose name is *Canva*, to artificially augment data by creating new photos with particular over imposed elements: texts of different colors, sizes and fonts, small forms and so on. The number of custom samples is the same for each artist to guarantee *the same 'treatment'* during training phase for each class.

The final result was represented by six groups of images, two for each considered class. These groups were collected in folders named as follows:

[1] "Crawler" (sometimes also called a "robot" or "spider") is a generic term for any program that is used to automatically discover and scan websites by following links from one webpage to another.

- CARAVAGGIO, MANET, VANGOGH: which contain photos found on the Internet. In particular, CARAVAGGIO counts 1749 items, MANET counts 2318 items and VANGOGH counts 4536 items
- CARAVAGGIO_CANVA, MANET_CANVA, VANGOGH_CANVA: which contain photos modified using Canva (56 for each artist)

The images vary widely in terms of size and shape, which is important for generalization.

It was obvious that the dataset collected was quite unbalanced (look at Figure 4). This observation was taken in account during the subsequent project steps.

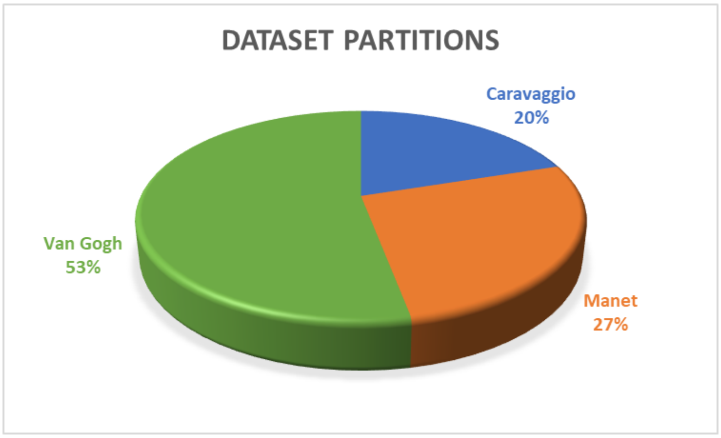


Figure 4. Dataset classes proportions

It is very important to observe that *the final dataset was actually obtained only after some trainings and validations* performed on the chosen basic neural network: InceptionResNetV2[2]. In fact, preliminary tests were able to show the presence of pictures very hard to classify correctly for several reasons and determined the need to delete/modify/add some pictures. For example:

- It was discovered thanks to misclassifications that the dataset still had some photos of paintings which were not works made by none of the considered artists and for this were difficult to classify. In particular, it was discovered that there was a painting made by Monet and not Manet, and that there was a painting not realized by Caravaggio but by an artist contemporary to him. Obviously, these samples were deleted
- Some samples representing particular paintings were misclassified in a large number of tests. For this reason, we decided to add to the dataset other samples that depicted those same paintings. The added samples chosen were *more qualitative* than the misclassified ones.
- Some misclassified samples had in common a background which occupied a large portion of the picture. Given that the network is actually thought to recognize samples in which the background (if present) occupies a small part, it was decided to crop properly these images

[2] In the following pages it is going to be clarified why transfer learning was chosen as technique to be used and why InceptionResNetV2 was selected as base model. The dataset split used for these preliminary tests is the same described in the section "Dataset splitting" of this report

Dataset Splitting

A fundamental task is represented by **dataset splitting** into two subgroups: *training and validation sets*. The first one is a data set of examples used during the learning process, while the second one is another set of samples used to provide an *unbiased evaluation* of the model fit on the training set. It is fundamental that training and validation sets are not related to each other in order to provide a good estimation of network performances.

Since the final private test set is balanced, we decided to obtain a balanced validation set too, even if the original collected dataset was unbalanced. This way, in fact, selecting a model with good performances on average across all classes would have been easier. In particular, the adopted procedure is the following one.

We realized a Python function, whose name is ***split_data***, which takes in input 4 parameters:

- *directories*: a list of folders, each of which containing samples of a specific class
- *destinations*: a list of folders. In particular two folders (one for train and the other for val) per class must be provided
- *val_perc*: an integer between 0 and 100, which represents the percentage of less numerous class that must be assigned to the validation set. The number of samples belonging to the other classes to assign to the validation set is equal to the number obtained from the percentage
- *num_classes*: the number of classes of the classification problem to deal with

First of all, *split_data* computes the number of samples available for each class, then identifies the **less numerous class** (which in our case is 'caravaggio'), computes its *val_perc*% (which in our case is 360, because we chose to impose **val_perc = 20**), extracts randomly from this same class a number *X* of samples corresponding to this percentage and assigns them to the validation set. The remaining part of this class is assigned to the training set.

After that, the function selects randomly *X* samples from each of the remaining classes (in our case 'manet', 'vangogh') and assigns them to the validation set. All the other samples are assigned to the training set.

In our case the final validation set is composed of 1080 images (360 for each class) and is completely balanced. It is important to observe that the followed procedure implies that inside validation set there are:

- 20% of total number of Caravaggio's samples
- 15.5% of total number of Manet's samples
- 8% of total number of Van Gogh's samples

Although 8% is not a high percentage, it can be assumed that it is a representative set of all Van Gogh's works for two reasons:

1. Samples that constitute this percentage were randomly selected
2. Van Gogh's style is quite homogeneous since there are no strong stylistic differences within his works

During splitting design we considered two further possibilities that we subsequently discarded for several reasons:

I. **Computing 10% of the most numerous class** in order to obtain a number X of samples to select from each class and to insert into the validation set. This approach would have implied that inside validation set there were:
    - 10% of total number of Van Gogh's samples
    - 19.5% of total number of Manet's samples
    - 26% of total number of Caravaggio's samples

This solution was discarded for two reasons:

1. Retrieving 26% of total number of Caravaggio’s samples and assigning them to the validation set implied that inside the training set only 74% of Caravaggio’s samples was left. Since ‘caravaggio’ was the less represented class inside the original dataset, we considered that this percentage was too small to train properly the network
2. The difference between the most numerous class’s percentage assigned to the validation set and the least numerous class’s percentage assigned to the validation set was larger than the approach we chose to use. In fact, |26 – 10| = 16, which is larger than |20 - 8| = 12

II. A standard splitting technique used when the dataset is unbalanced is ***stratified sampling***. If we had applied it, we would have obtained a validation set characterized by proportions similar to the original dataset ones (53% Van Gogh, 27% Manet, 20% Caravaggio). Since the final test set, instead, is balanced, we discarded this approach. In fact, it wouldn’t have given us the certainty to select a model with good performances on average across all classes

Pre-processing pipeline

First of all, it was considered that the obtained training set was quite **unbalanced**. Since our purpose was obtaining a model with *good performances on average across all classes* (in fact we knew that the final private test set that would have been used to evaluate our model was *perfectly balanced*) we had to choose *a strategy to balance the training set*.

An important observation to make is the following one: *resampling of data should always be done after splitting data into training and validation sets* in order to avoid any kind of correlation between them. In fact, it would negatively affect network performance esteem.

The technique we chose to use is **random oversampling***:* it consists of randomly selecting samples from the minority classes and adding copies of them to the training dataset to get the desired class distribution (in our case 33.3% for each class) before training. It is fundamental to observe that we decided also to use **on-the-fly image data augmentation**, which ensures that *throughout the training the network never sees the exact same sample more than once*. This is crucial to avoid overfitting as much as possible.

In particular, we created a function, whose name is ***balancer***, which provides a random oversampling of the original training set. *balancer* takes three parameters as input:

- *directories*: a list of folders, each of which containing samples of a specific class
- *destinations*: a list of folders, one for each class, meant to contain new samples, which are actually copies of available samples. If directories and destinations are the same, destinations of course contain also original samples
- *num_classes*: the number of classes of the classification problem to deal with

Oversampling implicitly ensures that all the batches provided to the network during the training are balanced, which is fundamental when (as in our case study) we want to give the exact same importance to all the classes.

It is significant to notice that random oversampling is not the only possible way of balancing an unbalanced training set. In fact, we took in account also other chances that we subsequently discarded for several reasons:

- *weight balancing*: Keras provides the possibility to set for every class a weight, which is then used during loss function calculation. This is very useful to give more relevance to samples from less populated classes: higher weights are assigned to them so that during training phase the mistakes made on them are considered more serious than the ones made on samples coming from more numerous classes. In our case, the weights assigned respectively to 'caravaggio', 'manet', 'vangogh' were 2.9, 2.1 and 1.0 and were easily computed dividing the number of samples of the most populated class ('vangogh') by the number of samples of each class. We made several tests but the obtained performances were not as good as the ones got using random oversampling. In particular:
    - the reached validation accuracy was a bit lower
    - the misclassified samples did not seem so hard to classify
    - there was a significant overfitting
- *undersampling*: it consists of selecting samples from the majority classes and deleting them from the training dataset to get the desired class distribution before training. We decided to avoid this technique because our purpose was training the network with as many samples as possible from every class
- *creating balanced batches from unbalanced data with a custom ImageDataGenerator*: technically it is possible to create a custom Generator that provides balanced batches to the network even if the training set is not balanced.
  It was decided not to further investigate this solution for the following reasons:
    - the available time for the completion of the project was not that much

- the expected performances with this approach were similar to the ones obtained with random oversampling

We decided to use *image data augmentation* to improve the performance and the ability of the model to generalize. Data augmentation is a famous technique to artificially create new training data from existing training samples, which is very useful mostly when the original dataset is not particularly big. Augmenting data grants a *broader exploration* of the input space and theoretically allows to have infinite samples to submit to the network. For the common case of images, **Keras** provides *ImageDataGenerator*, which is a class that can be used to automatically apply a set of image transformations with random parameters in predefined intervals. *ImageDataGenerator* is particularly useful because it provides the *flow_from_directory* method, which is used to load training data from the disk and to generate batches with *real-time augmentation* according to the parameters passed to the *ImageDataGenerator* constructor. It is important that the applied transformations do not affect the target/label of the images. Moreover, the transformations must be carefully chosen according to the particular classification problem to solve.

Besides, *flow_from_directory* provides a parameter (*target_size*) according to which **resizing** all the images read from disk. Given that the selected network architecture (presented in the next paragraph) has an image input size of **299-by-299**, every image was resized to these same dimensions.

The *flow_from_directory* method was particularly useful for us because it gave us the opportunity to load from the disk small sub-sets of data one after the other, instead of loading in ram the entire training set, which would have been impossible given its huge dimensions.

The *transformations that we chose to apply on training samples* are the following ones:

- **rescale = 1./255**: data *normalization* is an important step because it ensures that each parameter (pixel in this case) has a similar data distribution. This makes convergence faster while training the network. The applied rescaling technique sets the new range of each sub-pixel to [0-1], in fact the minimum and the maximum acceptable values for each sub-pixel are at the beginning 0 and 255
- **brightness_range = (0.3, 1.3)**: an important variable to consider when taking a picture (in particular when the subject is a painting) is the *environmental light condition*. In fact, photos depicting the same subject taken in different times of the day can be characterized by strong differences in terms of brightness. Since we wanted to make our network robust in respect to this, we applied this transformation choosing as bounding parameters values that, looking at the generated samples, seemed to be the most appropriate. Actually, values under 0.3 as lower limit could lead to too dark images, while values over 1.3 as upper limit could lead to too bright images
- **shear_range = 20**: since the model must be robust in respect to pictures acquired from different angles, we decided to add this kind of transformation. In fact, it applies a distortion along an axis in order to change the perception angles, which is convenient to simulate photos which depict the same painting from different perspectives. The bound was set to 20 because higher values would have implied too strong distortions
- **rotation_range = 10**: real life pictures are characterized by the fact that they are not perfectly horizontally aligned, for this reason considering rotation as a possible transformation is fundamental. Moreover, coupling rotation and shear ensures that a single original image can be seen by the network during the training phase according to a lot of different points of view, which makes the model extremely robust with regard to angles and perspectives. We chose 10 as bound because higher values could lead to unrealistically tilted samples
- **horizontal_flip**: given that pictures can be sometimes mirrored (depending on camera settings) we opted for horizontal flip as another possible transformation

- **fill_mode = 'reflect'**: when applying transformations such as rotations and shears the generated image can show some gaps which must be filled choosing a particular filling strategy. The most appropriate one in our case is represented by 'reflect' because all the others imply not realistic generated samples; 'reflect', instead, gives life to images which do not risk to confuse the network because the painting is not excessively adulterated.

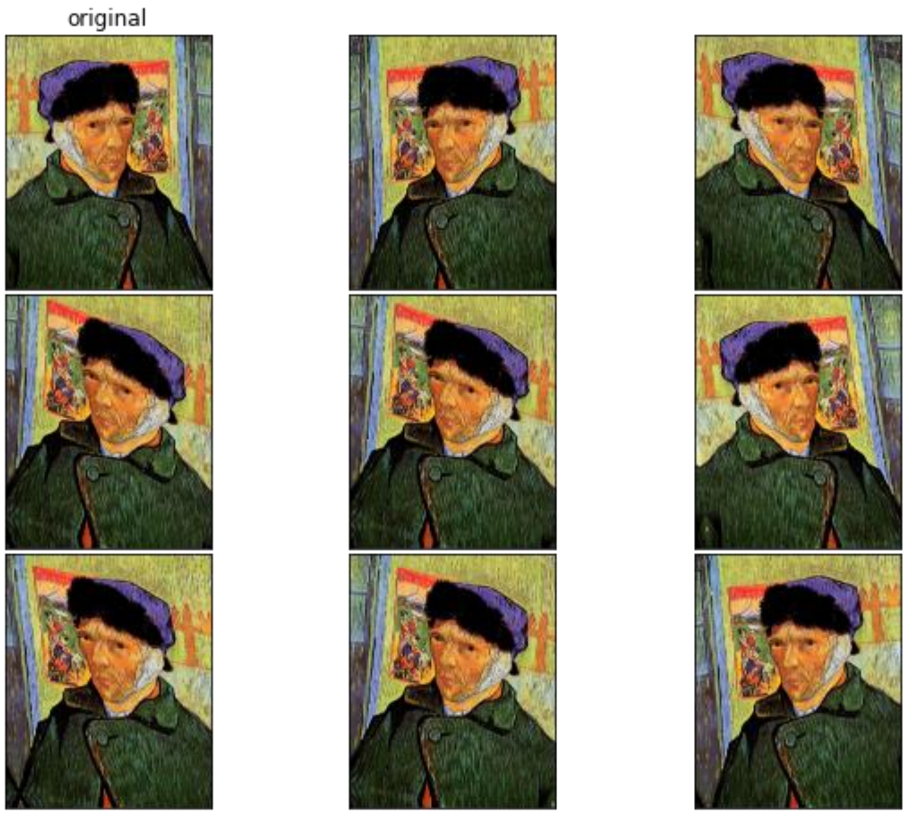


Figure 5. An example of samples generated by the chosen data augmentation settings

The *transformations applied on validation samples* are just:

- **rescale = 1./255**
- **resizing to 299x299**

They are fundamental to exploit at most what the model learned during the training phase. In fact, a different pre-processing in terms of rescaling and resizing of validation (or test) images could lead to inaccuracies in the evaluation of network performance.

Another important aspect to note is that the **randomness** with which ImageDataGenerator applies the transformations helps in general to **avoid overfitting**.

Network Architecture

The technique we selected is **transfer learning** because starting from a pre-trained model is a great advantage as it allows us to have a very deep network already trained on a vast dataset. Our aim was to choose an appropriate existing network and to adapt it to our purposes by making small changes to its final part. In most cases the neural networks used for transfer learning are trained on *ImageNet* (a very large dataset characterized by 1000 different classes) and, therefore, their last layer is made up of 1000 neurons, one for each class of ImageNet. In light of this, it was necessary to replace the last layer of the chosen pre-trained network with a layer made up of just 3 neurons because our classification problem presented just three classes ('caravaggio', 'manet', vangogh').

To select the most appropriate base model we had to make a lot of tests, alternating trainings and validations and using the dataset splitting previously discussed. In particular, among the others, we tested *Xception*, *VGG16*, *ResNet50*, *ResNet152*, *InceptionV3*, different versions of *EfficientNet* and *InceptionResNetV2*. The latter resulted to be the best one in terms of reached validation accuracy.

Model	Size (MB)	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth	Time (ms) per inference step (CPU)	Time (ms) per inference step (GPU)
InceptionResNetV2	215	80.3%	95.3%	55.9M	449	130.2	10.0

Table 1. InceptionResNetV2 characteristics (source: https://keras.io/api/applications/)

InceptionResNetV2 is a non-sequential convolutional neural network trained on ImageNet and characterized by skip connections, normalization layers, blocks with several parallel paths whose outputs are concatenated, which are all useful strategies to reduce vanishing gradient problem without reducing the network length.

Given that *InceptionResNetV2* has an image input size of 299-by-299, we had to adapt all our photos to these dimensions by providing to *flow_from_directory* method the right *target_size* parameter: (299, 299).

The reason why transfer learning is so efficient is that the first layers of a good pre-trained network are already able to extract pretty general features that are common to the most of image classification problems, such as shapes, angles, colors, etc. The last layers, instead, are the ones that must be trained more intensely on problem specific data, in fact they have to learn how to extract features related to the specific problem to solve. Moreover, in our specific case, using a network previously trained on ImageNet was particularly useful also because all the investigated artists usually painted lifelike scenes where there were objects that we expect to find in ImageNet.

We even tried to add to the base network other layers (such as convolutional and pooling ones) but without obtaining significant improvements. Moreover, since increasing the complexity of an already very complex network risks to produce overfitting (mostly when the available data is not that much) we decided to abandon this idea.

Training Hyperparameters

Given that we had to do with a multi-class classification problem, we decided to use:

- **Softmax as activation function for the output layer**
  for the k-th component of the output:
  $$y_k = f(net_k) = \frac{e^{net_k}}{\sum_{j=0}^{s-1} e^{net_j}}$$
  It guarantees that the output of each last layer neuron is between 0 and 1 and that the sum of all last layer neurons outputs is equal to 1. This means that the outputs of the neurons which are part of the output layer can be interpreted as probabilities

- **Categorical cross-entropy as loss function**
  $$J(y,t) = -\sum_{j=0}^{s-1} t_j \log y_j$$
  which can be seen as a generalization of binary cross-entropy function, that is instead commonly used as loss function in binary classification problems

After few tests we chose **Adam as optimization algorithm**. We explored also other optimizers such as AdaGrad, RMSProp, SGD with momentum, etc. but the best performances were obtained with Adam. In particular, according to its authors, Adam is able to combine the advantages of AdaGrad and RMSProp and is empirically one of the best algorithms for learning optimization.

Regarding **batch size**, we made several tests and realized that the best solution for us was **32**.

Besides, it is important to remember that when calling the method *fit* (or *fit_generator*) on a model with a generator (as we did) Keras has no way of knowing when an epoch is finished because you are not passing the whole training set at once. For this reason, it is compulsory to also provide the *steps_per_epochs* parameter in order to specify the number of minibatches which constitute a single epoch. Given that in general at the end of every epoch a lot of significant measures (such as accuracy) are performed, and some callbacks could be called, deciding a good epoch size is crucial to determine a correct course of the training. In our case, we set the epoch size equal to the total number of training samples (obtained after oversampling) divided by the batch size. This guaranteed that a single epoch lasted no more than 5-6 minutes, so in case of problems related to the machine or to Colab the amount of “wasted work” would not have been that big.

Choosing the right moment to stop the training in order to avoid overfitting is crucial for obtaining an acceptable model that is capable of generalizing. A good technique that can be easily implemented in Keras to do that is represented by the **Early Stopping**, which can be considered a *particular form of regularization* (one of the typical ways through which overfitting can be avoided) and consists of monitoring a network parameter *X* and stopping the learning procedure when *X* does not increment anymore after *n* epochs. Obviously, *X* and *n* are hyper-parameters, which means that their choice can affect training and the performances of the final obtained model. As *X* we chose the **validation accuracy** because it gives in general a very easy and immediate overview of training, while as *n* we chose 3 (this parameter in Keras is called *patience*) because it was neither too high nor too low so it could ensure overfitting limitation without the danger of stopping too early. In fact, it is fundamental to remember that the loss function in general has not a single minimum and its trend can sometimes change. Moreover, it is very useful to save the best weights found so far (even if the learning procedure keeps going): to do that in Keras it is sufficient to use the parameter **ModelCheckpoint**.

The chosen strategy for the training is the following one.

We decided to split the training in 3 different phases

- ***PHASE 1***
  - Only the fully connected last layer is trainable
  - The maximum number of epochs is set to 30
  - Learning rate = 10^{-3}
  - The other Adam parameters are set to default values (β_1=0.9, β_2=0.999)
- ***PHASE 2***
  - All the possible trainable weights are set as trainable
  - The maximum number of epochs is set to 20
  - Learning rate = 10^{-4}
  - The other Adam parameters are set to default values (β_1=0.9, β_2=0.999)
- ***PHASE 3***
  - All the possible trainable weights are set as trainable
  - The maximum number of epochs is set to 20
  - Learning rate = 10^{-5}
  - The other Adam parameters are set to default values (β_1=0.9, β_2=0.999)

The technique to which the applied strategy is inspired is called fine-tuning[3]**.**

It is fundamental that *the unfreezing of the whole model is performed after that the model with frozen layers has been trained to convergence* (at the end of PHASE 1). In fact, if you mix randomly-initialized trainable layers with trainable layers that hold pre-trained features, the randomly-initialized layers will cause very large gradient updates during training, which will destroy your pre-trained features. Another thing to notice is that the unfreezing of the whole model is accompanied by a **significant reduction of learning rate** (PHASE 2 and even more PHASE 3). This is essential to avoid (or reduce) overfitting and to readapt in a slow and incremental way the pretrained weights on the available dataset.

The choice of this particular training strategy was in part inspired by this paper, which also deals with the problem of artist identification.

[3] An interesting guide can be found at this link: https://keras.io/guides/transfer_learning/

Performance Analysis

After the training process we had to evaluate our network performances by using some appropriate metrics such as validation accuracy, precision, recall etc. The following paragraph is focused on the performance analysis of our best model.

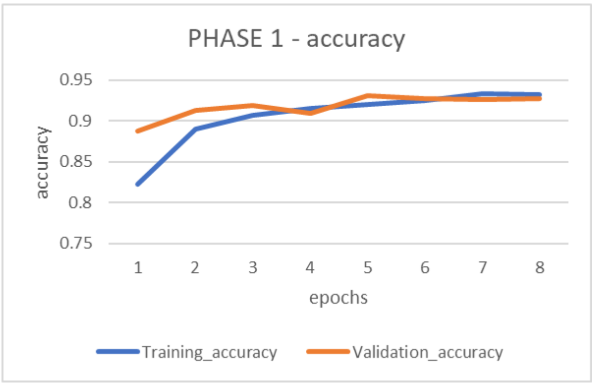


Figure 6. PHASE 1 - accuracy trend

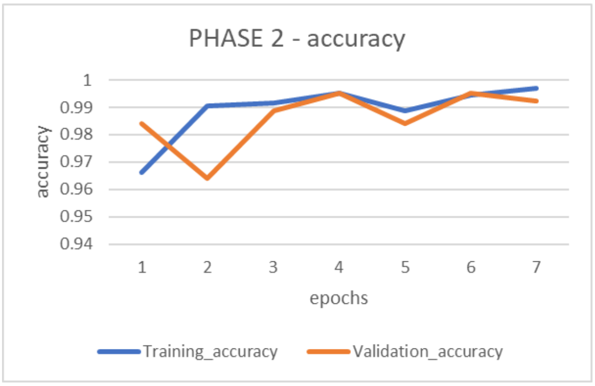


Figure 7. PHASE 2 - accuracy trend

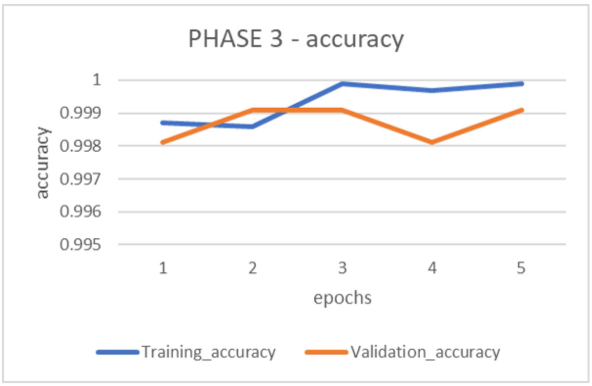


Figure 8. PHASE 3 - accuracy trend

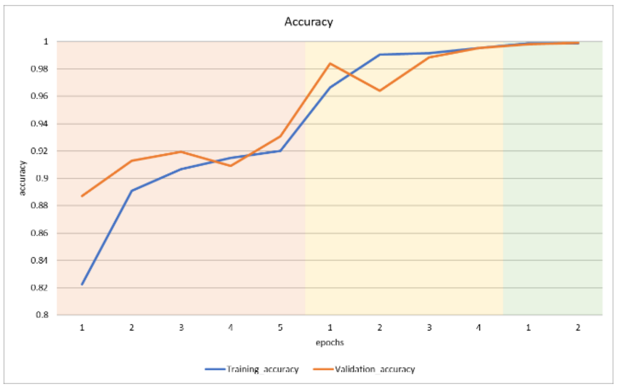


Figure 9. Accuracy trend

The previous images show how the **training and validation accuracies** of our network change during the training. It must be noted that the last three epochs of PHASE 1, PHASE 2 and PHASE 3 did not contribute to the final network weights because of the *patience* parameter with which the network was trained. This is the reason why these epochs are not reported in Figure 9. It is particularly interesting to observe that between PHASE 1 and PHASE 2 there was a *noticeable jump for both training and validation accuracies*, which actually corresponds to a *learning rate decrease by a factor of 10* and to the fact that *the whole network was allowed to change its weights*. The jump indicates that the network started to adapt better to our dataset. Another *smaller jump for both training and validation accuracies* can be seen between PHASE 2 and PHASE 3 and it corresponds to another *decrease of the learning rate by a factor of 10*. This further jump indicates that there was a further, although small, improvement of the network.

It can be noted that there is not a significant gap between training and validation accuracies, which means that there is *no overfitting*. The network, in fact, is expected to be able to *generalize*, that is to say it manages to correctly classify paintings by Caravaggio, Manet and Van Gogh which had never seen before because not included in the training set.

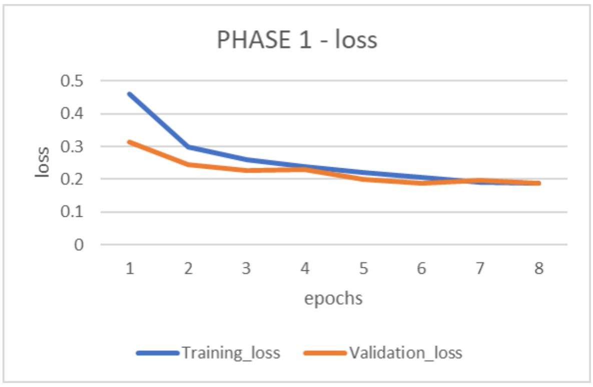


Figure 10. PHASE 1 - loss trend

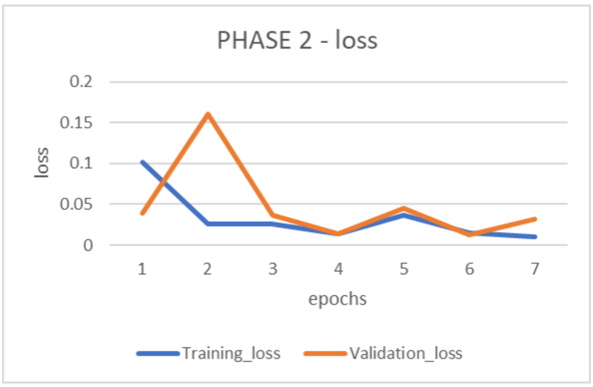


Figure 11. PHASE 2 - loss trend

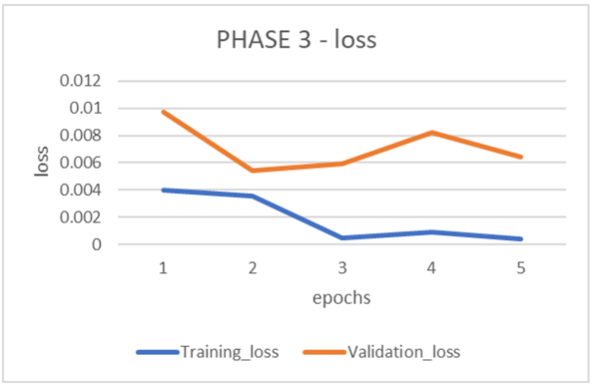


Figure 12. PHASE 3 - loss trend

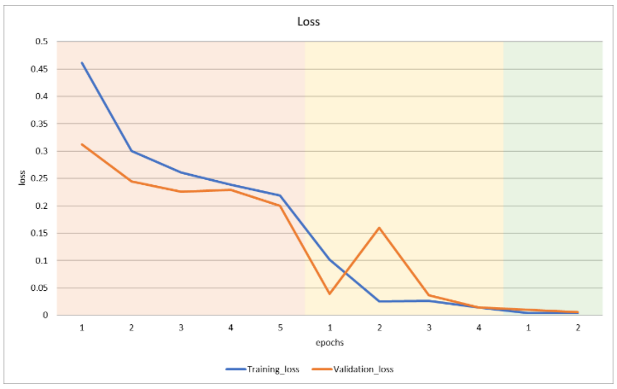
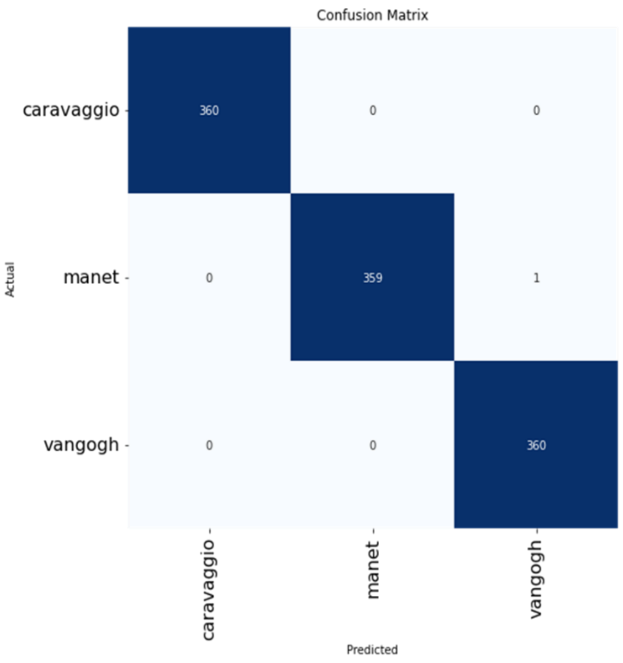


Figure 13. Loss trend

A similar analysis as the one conducted for the accuracy can be done also for the **loss function**. In particular, the loss function trend shows *two downward jumps* (one more significant than the other) corresponding to the beginning of PHASE 2 and PHASE 3.



	caravaggio	manet	vangogh
caravaggio	360	0	0
manet	0	359	1
vangogh	0	0	360

Figure 14. Confusion matrix

Another tool we relied on during performance analysis was the **confusion matrix** (Figure 14). Each row of the matrix represents the instances in an actual class while each column represents the instances in a predicted class. In our case the maximum possible value for a cell is 360, because there are 360 images per artist in our final validation set. *We performed several tests on different models before obtaining the best one*. After each test, we always used the confusion matrix to understand which were the critical issues. In most of the cases the models assigned certain paintings by Van Gogh to Manet, and vice versa, while there were no particular problems with Caravaggio's works. This can be easily explained in the light of the fact that Van Gogh and Manet's styles are pretty similar since they were part of related artistic currents as Impressionism and Post-Impressionism. Moreover, they usually depicted similar subjects, as still lives, portraits and landscapes.

We managed to tackle this problem *by trying different pre-trained models* and *adding to the dataset other versions of the paintings which were more often misclassified*. This led us to the final dataset and made us realize that the best pre-trained network for our problem was InceptionResNetV2.

Our best model showed outstanding performances. In particular, only one painting (Figure 15) resulted to be misclassified. Its name is *"Oloron Sainte Marie"* and was assigned by the network to Van Gogh even if its author is Manet. The reason why this happened is probably that the photo is characterized by colors which are slightly different by the ones actually used by Manet (Figure 16), just as a sort of yellow light filter had been applied to the picture. Since yellow is one of the most predominant colors in Van Gogh's works, it is understandable that the network had made this mistake.

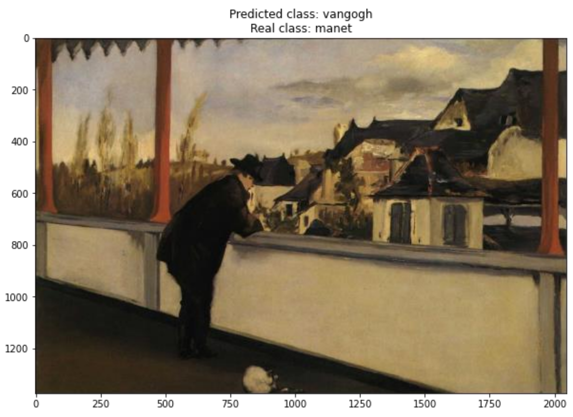


Figure 15. Misclassified sample

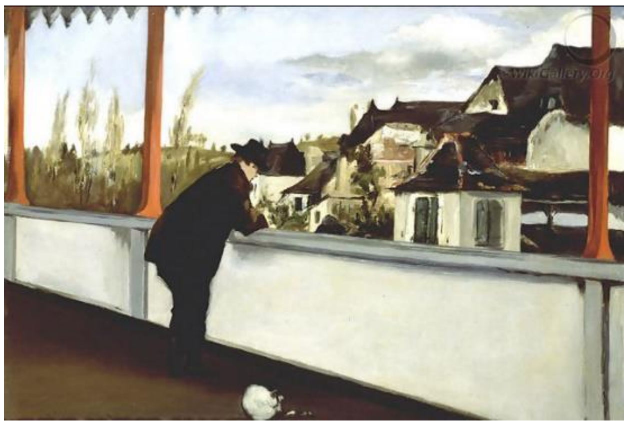


Figure 16. Original painting by Manet

Other useful metrics to evaluate a classification model are the following ones:

- **Precision**: it is intuitively the ability of the classifier not to label as positive a sample that is negative. It is defined as:

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives}$$

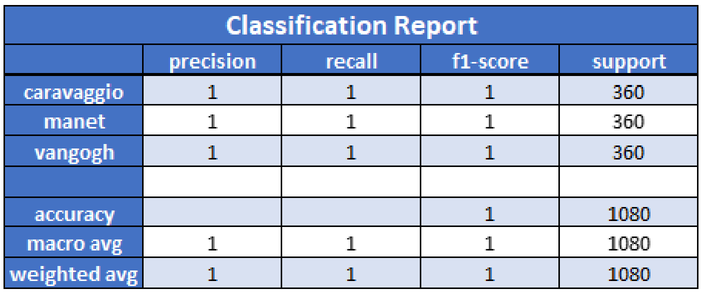
- **Recall**: it is intuitively the ability of the classifier to find all the positive samples. It is defined as:

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives}$$

- **F1-score**: By definition, F1-score is the harmonic mean of precision and recall, defined as:

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall}$$

The results reached by the network in terms of these metrics (reported in the table below) are literally ideal thanks to the fact that only one mistake was made on the validation set.



Classification Report				
	precision	recall	f1-score	support
caravaggio	1	1	1	360
manet	1	1	1	360
vangogh	1	1	1	360
accuracy			1	1080
macro avg	1	1	1	1080
weighted avg	1	1	1	1080

Table 2. Classification report

Another metric that we considered was **top-2 classification accuracy**, which assumes that a painting is correctly classified if the correct artist is in the top 2 highest scores generated by the network. Our best model reached 100% top-2 classification accuracy.

Further analysis

In our qualitative analysis we chose the following images as representative samples:

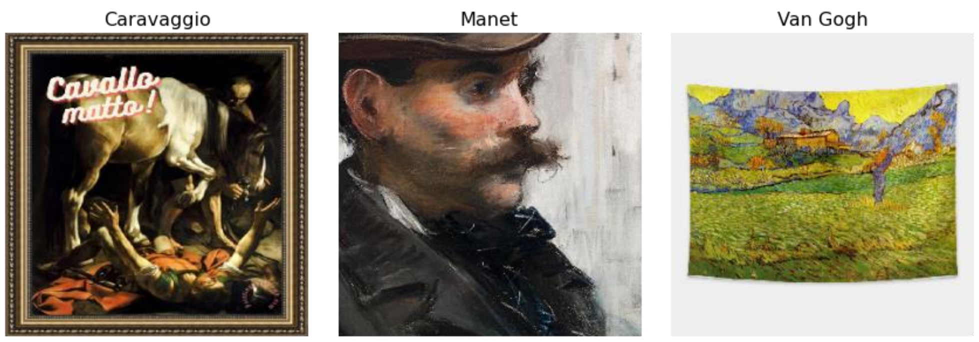


Figure 17. Samples chosen for qualitative analysis

Our purpose was understanding how the network analyzes the images in order to extract features and select the most sensitive information. The analysis, in particular, was carried out using *"attention maps" (such as Saliency, GradCam, etc.)*, which are able to *visualize which pixels/portions in an image contribute most to the predicted score for that image*. To achieve that, we decided to analyze three very representative pictures:

- The first one, from 'caravaggio', has multiple subjects (a horse and some people) and presents a *sharp contrast between the black background and the foreground*, making the outline of the subjects more visible. Also, two *disturbance elements* are present: a frame and a colored writing occluding the picture underneath
- The second one, from 'manet', is a simple detailed view of a *human subject* which focalizes on his face, his clothes, and a reduced background
- The third one, from 'vangogh', is the *printing of a painting* with a consistent white background. There are some distortions due to the folds of the cloth on which the landscape is printed on

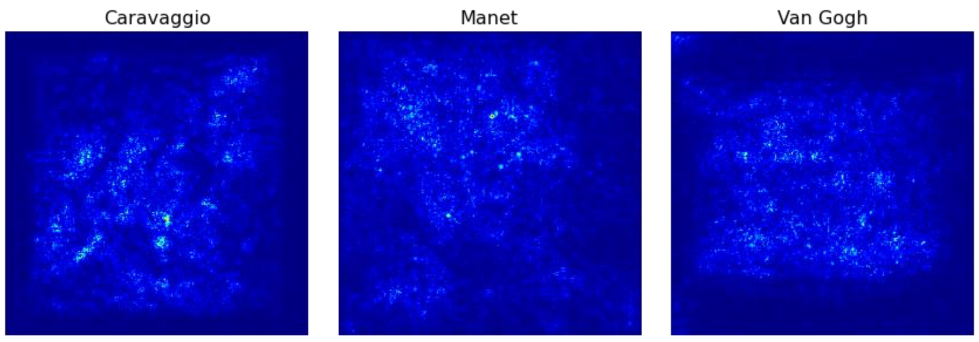


Figure 18. Saliency maps of the chosen images

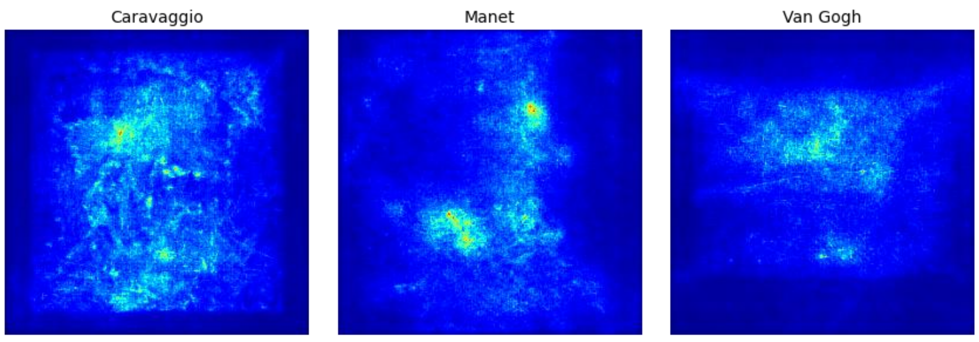


Figure 19. SmoothGrad maps of the chosen images

The previous images (Figure 18 and Figure 19) are respectively the Saliency maps and the SmoothGrad maps of the selected pictures. *The latter has the same behavior of the former with some emphasized highlights thanks to the addition of a particular noise pattern*. Analyzing each picture, we can observe some intuitively predictable behaviors.

In the painting from ‘caravaggio’ the “hottest zones” are the ones *related to the subjects*; most of the attention is directed towards the horse, although the human subjects are clearly distinct from the background too, especially in the SmoothGrad case. Another apparent distinction in both kind of maps is the sharp reduction in focus intensity going from the painting area to the pixels representing the frame.

The same thing can be observed in the ‘vangogh’ picture, where the *outline of the printing is immediately visible* in both the Saliency and the SmoothGrad maps. In this picture the highlighted details are the ones from the most inhomogeneous areas of the painting, mainly the houses and some of the mountains. The green field in the lower part of the image, with way less color variations, appears to be less relevant to the network.

In the ‘manet’ picture the distinction between the subject and the background is less pronounced but still present; the network mainly focuses on the upper part of the subject’s face and on some details on his clothes.

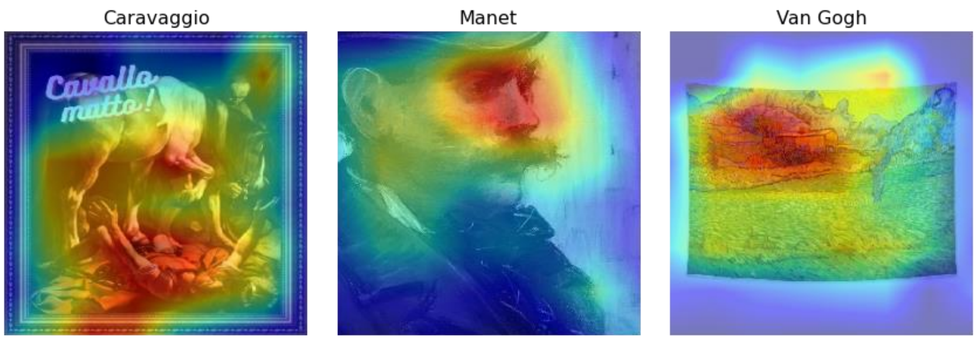


Figure 20. GradCam maps of the chosen images

The previous images (Figure 20) are the heatmaps obtained through GradCam of the same three pictures. They represent another way of visualizing how the network evaluates different zones in the paintings, assigning them a certain weight. The behavior depicted here is pretty similar to the one which is evident from the Saliency and SmoothGrad maps, but there are some interesting differences: in the first picture the focus is slightly shifted, and the human subjects appear to be more relevant for the network while the horse is not as significant as before. Also, some details of the painting frame are now considered; the colored writing, though, remains overlooked.

In the second picture, the heatmaps shows a stronger focus on the subject’s face (in particular eyes and nose) and less attention to the clothes.

In the third picture the focus remains distributed in the upper part of the painting, where houses, plants and mountains are located and a greater amount of fine details is present. As before, the repeating pattern of grass in the field is partially ignored.

Conclusions

Our best model results and the qualitative analysis conducted in the previous paragraph demonstrate that complex Convolutional Neural Network (as InceptionResNetV2) are a powerful and useful tool for the problem of artist identification: it means that *a properly trained deep learning network is able to learn how to recognize artists' styles*. Obviously, it would be very interesting to collect an even larger dataset (including also other artists) and to obtain a very powerful model which would manage to classify works made by tens or even more painters.