UNIVERSITÀ DI PISA

Artificial Intelligence and Data Engineering
Computer Intelligence and Deep Learning

Artist Identification from Paintings

Tommaso Amarante, Edoardo Morucci, Enrico Nello

Github repository:
https://github.com/TommyTheHuman/CIDL-ArtistClassification

Contents

Abstract

The creation of art is a profound expression of human imagination and intellect. Our ability to communicate and express our imagination sets us apart from other species.
We have come a long way, and now computers are creating paintings as well. For art enthusiasts, identifying the work of their favorite artists through years of study and research is not difficult. They can easily recognize a painting as being created by a particular artist they admire. But can a computer do the same? Can a machine accurately identify the artist behind a painting?

Chapter 1

Introduction

The aim of the project is to create an algorithm that can accurately identify the artist of a painting using state-of-the-art precision.
In particular, we show how to utilize Convolutional Neural Networks (CNNs) for identifying the artist of a painting. Various techniques will be employed to model the CNNs, including building the networks from scratch, utilizing pre-trained networks, and creating an ensemble network composed of the most effective classifiers.
We chose to study in deep this field because it could be exploited for examples in museums that tries to modernize themselves through digitalization. For instance, nowadays many museums use tech guides to help visitors (e.g. the Louvre in Paris) during the exhibition. We thought that our model prediction could be useful to dynamically obtain information about the paintings and the authors by simply taking a picture of the painting during the visit.

1.1 State of the art

Artist identification has traditionally been a task performed by human experts, such as the Artsy's Art Genome Project, which is led by specialists who categorize art by hand. Despite its high accuracy in classification, this approach is not very efficient. Then, the first efforts to use machine learning for this task have mostly focused on identifying key features that differentiate artists and styles. Various image features have been utilized, such as scale-invariant feature transforms (SIFT), histograms of oriented gradients (HOG), and others, with a particular emphasis on recognizing styles in fine-art paintings [1].

The very first significant attempt to address artist identification through machine learning was made by J. Jou and S. Agrawal in 2011 [2]. They tested several multi-class classification techniques, including Naive Bayes, Linear Discriminant Analysis, Logistic Regression, K-Means, and SVMs, and achieved a maximum accuracy rate of 65% for identifying an unknown painting across five artists. The problem was later tackled again in the Rijksmuseum Challenge [3]. The Rijksmuseum Challenge aimed to accurately predict four key attributes of 112,039 photographic reproductions of artworks exhibited in the Rijksmuseum in Amsterdam: the artist, type, material, and creation year. The challenge was divided into four separate tasks, with the artist classification task achieving a test accuracy of around 60%.

One year later, Saleh and Elgammal in their paper [4] attempted to identify artists using a large and diverse dataset. They used images of 81,449 fine-art paintings from 1,119 artists spanning fifteen centuries, from ancient to contemporary, and achieved an accuracy rate of 59% using generic features (in the paper they tried to use also CNN, but reaching only an accuracy of 33.62%).

More recent efforts include the Painter by Numbers, a prediction competition organized by Kaggle [5]. This competition employed a pairwise comparison approach, where participants were asked to develop an algorithm that could examine two images and determine if they were created by the same artist. While this competition was not focused on the same objective as the Rijksmuseum Challenge, it can be considered the first application of deep learning to the problem of artist identification.

In 2017, Nitin Viswanathan made significant progress in the field of artist identification [6]. Using the same dataset from the Kaggle Challenge, he proposed the use of ResNet with transfer learning. He first kept the base ResNet weights constant and only updated the fully-connected layer for a few epochs. This trained network achieved a training accuracy of 0.973 and a testing accuracy of 0.898.

1.2 Dataset

The dataset we chose is *Best Artworks of All Time*, that was created by was scraping from *artchallenge.ru* during the end of February 2019 [7]. It contains a collection of artworks of the 50 most influential artists of all time, together with some basic information retrieved from wikipedia. The total size is about 2GB of data and about 8k unique images.
The dataset contains three files:

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

Specifically, the data are organized as follows:

```
/
└── images
    ├── images
    │   ├── Albrecht Durer
    │   ├── Alfred Sisley
    │   ├── Amedeo Modigliani
    │   └── Other 47 artists
    ├── resized
    └── artist.csv
```

Below are shown some samples from three artists inside the *images* folder:

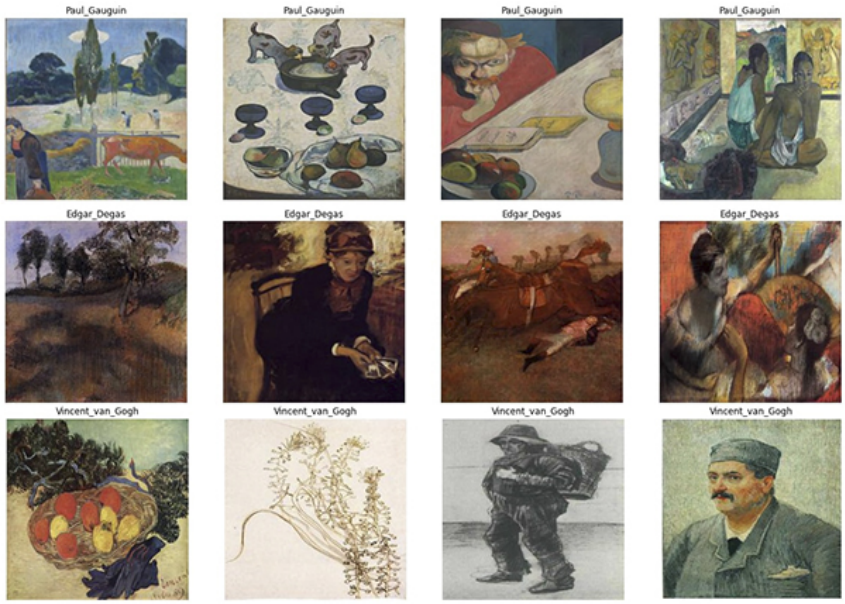


Figure 1.1: Some paintings from three artists inside *images* folder

To make the dataset more manageable for learning, the number of classes was reduced by only including artists who had created at least 200 pieces. Despite this, the dataset remained unbalanced, with some artists having many more pieces represented than others. To address this, class weights will be calculated and used during the fitting process.

We then created the train, validation and test sets using the *image_dataset_from_directory* function provided by Keras, setting 70% - 15% - 15% as percentages respectively. This selection of images was made with consideration for only the artists who had at least 200 paintings.
The resulting Learning Set is organized as follows:

- **Training set**: 3004 files belonging to 11 classes.
- **Validation set**: 656 files belonging to 11 classes.
- **Test set**: 639 files belonging to 11 classes.

Hence, we have a total of 4299 different pictures.

Chapter 2

Data Preprocessing

2.1 Image Resizing

We decided to resize images to 224x224 px in order to get a manageable size of dataset easy handable in Colab, as most of the original ones were scraped in HD (800x600) from *artchallenge.ru*.

```
def resizeimages(outputdir, newwidth, newheight):
  # Iterate over the input images

  for artist in tqdm(CLASSES):

    artistdir = os.path.join(BASEDIR, artist)
    for strimage in os.listdir(artistdir):

      imagepath = os.path.join(artistdir, strimage)
      # Extract the filename from the path and save the resized image
      if not os.path.exists(outputdir + "/" + artist):

        # if the demofolder directory is not present
        # then create it.
        os.makedirs(outputdir + "/" + artist)


      # Read the image and resize it
      image = Image.open(imagepath)
      image = image.resize((newwidth, newheight), Image.ANTIALIAS)

      image.save(outputdir + "/" + artist + "/" + strimage)

# Define the paths to the input and output directories
outputdir = "images-resized"

# Define the new width and height for the images
newwidth = 224
newheight = 224

# Resize the images
```

```
resizeimages(outputdir, newwidth, newheight)
```

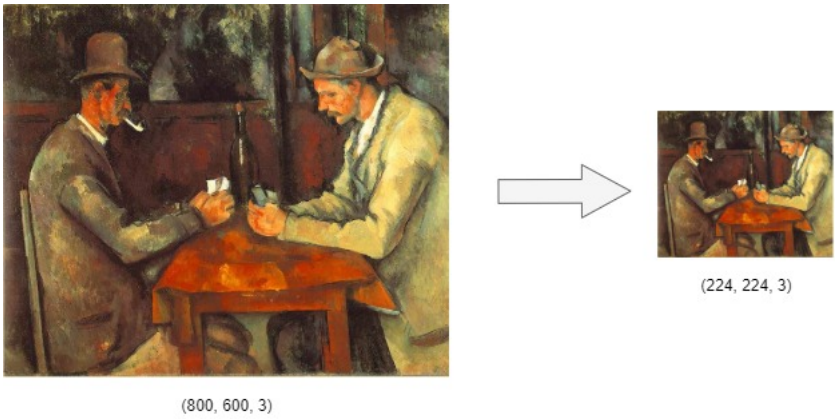


Figure 2.1: original image resizing

2.2 Data Balancing

The original dataset was heavily unbalanced as shown in the following plots:

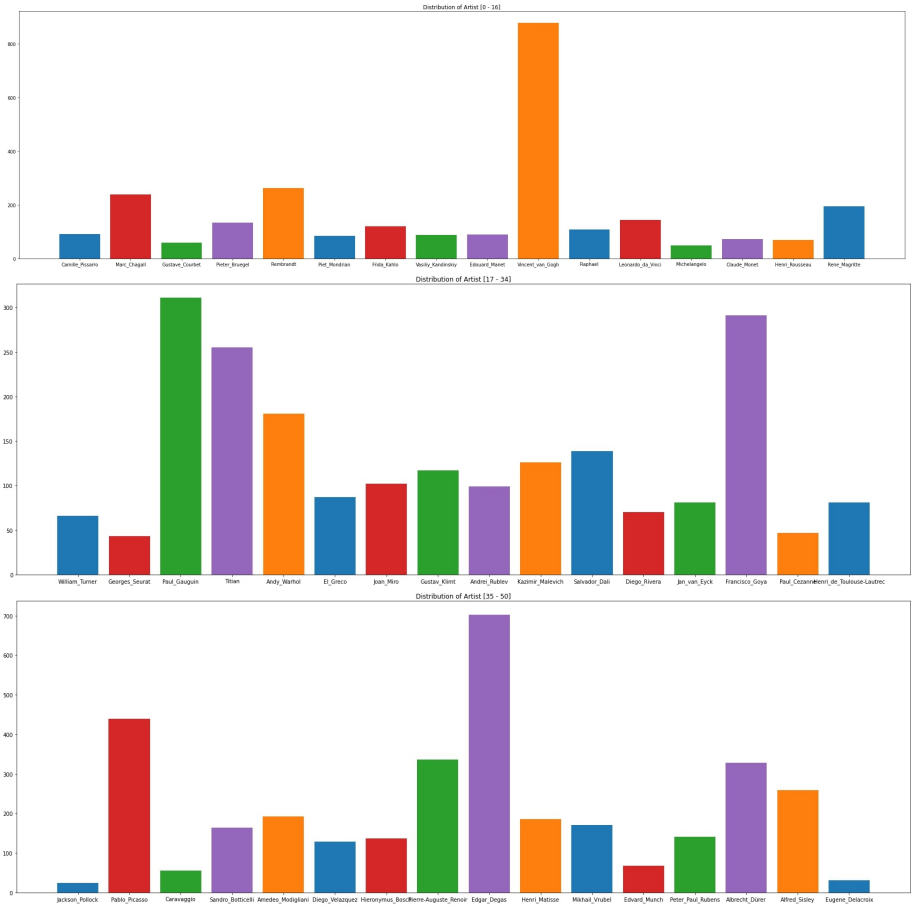


Figure 2.2: class cardinality distribution highlights heavily unbalanced dataset

We can notice that some artist, e.g., Van Gogh's paintings are 877 against the 31 of Eugene Delacroix's. Thus, as mentioned in paragraph 1.2 we selected only the artists with at least 200 pieces and discarding the others.
This operation was done to reduce the number of classes to a number per which the ratio between the number of artists and images was reasonable for learning. In doing so, we reduced the number of artists (classes) from 50 to 11:

```
CLASS = []
for key in classcardinality:
  if classcardinality[key] ¿ 200:
    CLASS.append(key)

newclasscardinality = -

for artist in CLASS:
  newclasscardinality[artist] = classcardinality[artist]
```

```
newclasscardinality

-'MarcChagall': 239,
 'Rembrandt': 262,
 'VincentvanGogh': 877,
 'PaulGauguin': 311,
 'Titian': 255,
 'FranciscoGoya': 291,
 'PabloPicasso': 439,
 'Pierre-AugusteRenoir': 336,
 'EdgarDegas': 702,
 'AlbrechtDürer': 328,
 'AlfredSisley': 259
```

Nevertheless, the dataset remained unbalanced:

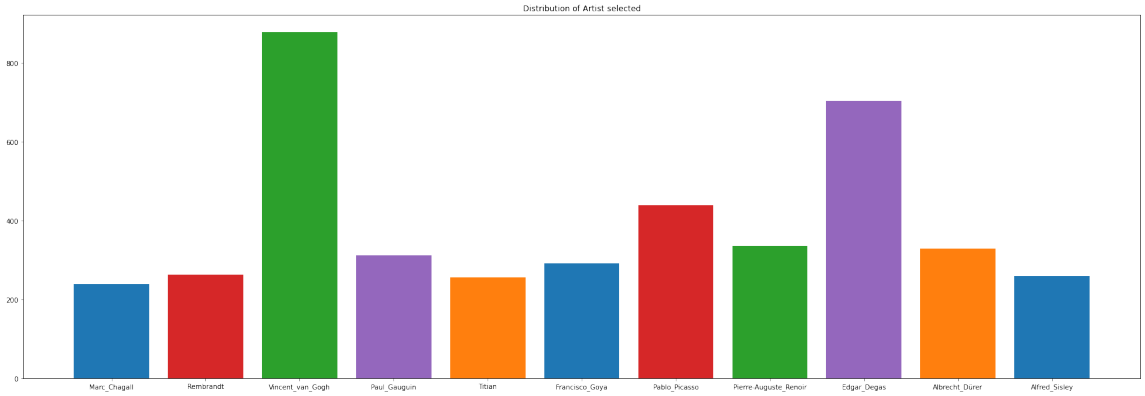


Figure 2.3: class cardinality for the remaining 11 artists

We can still notice the gap between 877 Van Gogh's paintings against the 239 of Chagall's. At this point, we decided to investigate two different approaches for addressing the issue of balancing data:

- **Data Augmentation**
- **Class Weight**

2.2.1 Data Augmentation

Data augmentation is a technique used to create more training data by applying random transformations to existing samples. This is done to ensure that the model is exposed to various aspects of the data during training, which can improve its ability to generalize. We decided to exploit data augmentation to rebalance our dataset; specifically, we equalize the number of paintings (images) for each artist to 500, only for the training set (after splitting the dataset). In Keras, data augmentation can be applied by adding specific layers at the beginning of the model.
The following transformations were used to generate new images:

```
def dataaugmentation(image, basepath, counter):
  seq = ks.Sequential([
      ks.Fliplr(0.5), # flip image with 50% probability
      ks.Sometimes(0.5, ks.Rotate(90)), # rotate image by 90 with 50%
       ↪  probability
      ks.GaussianBlur(sigma=(0, 1.5)) # apply gaussian blur with
       ↪  sigma between 0 and 1.5
  ])
```

```
  # Apply trasformation to image
  imageaug = seq.augmentimage(image)

  filename = str(counter) + ".jpg"

  Image.fromarray(imageaug).save(os.path.join(basepath, filename))

CARDINALITYTHRESHOLD = 500
for artist in tqdm(CLASSES):
  dir = os.path.join(TRAINPATH, artist)
  if len(os.listdir(dir)) ¡ CARDINALITYTHRESHOLD:
    initiallen = len(os.listdir(dir))
    counter = 0
    while len(os.listdir(dir)) ¡ CARDINALITYTHRESHOLD:
      imagelist = os.listdir(dir)
      index = random.randrange(0, initiallen)
      imagepath = os.path.join(dir, imagelist[index])
      image = Image.open(imagepath)
      image = np.array(image)
      dataaugmentation(image, dir, counter)
      counter += 1
```

Figure 2.4: Data Augmentation trasformations

We decided not to introduce too many images transformation as we deal with paintings and for the networks to better recognize the style of different authors we should not alter the lines/colours of the original depictions.

2.2.2 Class Weights

To tackle the unbalancing issue, the other attempt we tried was to leave the dataset unbalanced, but this time we added the class-weights to each artist (class) in a proportional way, e.g., the number of his paintings with respect to the total paintings:

$$class\_weight = \frac{Total\#ofpaintings}{Total\#ofclasses \cdot Total\#ofpaintingsofcurrentauthor}$$

We introduce class weights as we know that due to the cardinality differences between class, the algorithms tend to get biased towards the majority values present and don't perform well on the minority values. This difference in class frequencies affects the overall predictability of the model and the algorithm will not have enough data to learn the patterns present in the minority class.

The effects of class weights results in forcing the training algorithm to take into account the skewed distribution of the classes. This can be achieved by giving different weights to both the majority and minority classes. The difference in weights will influence the classification of the classes during the training phase. The whole purpose is to penalize the misclassification made

by the minority class by setting a higher class weight and at the same time reducing weight for the majority class. Thus, during the training we give more weightage to the minority class in the cost function of the algorithm so that it could provide a higher penalty to the minority class and the algorithm could focus on reducing the errors for the minority class. So, the original Cross Entropy that we chose as our Loss Function:

$$H_y(y') = -\sum_{i=1}\sum_{k=1}^{K} y_{i,k}\log(y'_{i,k})$$

will be weighted as so:

$$H_y(y') = -\sum_{i=1}\sum_{k=1}^{K} w_k y_{i,k}\log(y'_{i,k})$$

For example, let *uc* be the Under-Represented class. By setting $w_{uc} = 10$ and $w_k = 1_{(k \neq uc)}$, we are essentially telling the model that miss-classifying 1 member from *uc* is as punishable as miss-classifying 10 members from other classes. This is roughly equivalent to increasing the ratio of class *uc* 10 times in the training set.
Obviously, the metrics to be taken into account at the end to check the performance of the model will be the F1 score, not accuracy.

Here is reported the snippet of code to implement the class-weight:

```
  GENERALPATH="/content/.../Data/Split-dataset/train"

classcardinality=-
for it in os.scandir(GENERALPATH):
    if it.isdir():
      classcardinality[it.name]=len(os.listdir(it.path))

df = pd.DataFrame(classcardinality.items(), columns=['Name',
 ↪  'NumPaintings'])
df["ClassWeight"] = df.NumPaintings.sum() /
 ↪  (len(CLASSES)*df.NumPaintings)
CLASSWEIGHTS = df['ClassWeight'].todict()
CLASSWEIGHTS

-0: 1.258483452031839,
 1: 0.556193297537493,
 2: 0.44549903603737206,
 3: 1.1925367209210005,
 4: 0.889546935149541,
 5: 1.5342185903983656,
 6: 1.3452754142409316,
 7: 1.6352749047359825,
 8: 1.1620889748549323,
 9: 1.508789552988448,
 10: 1.4923000496770988
```

2.2.3 Data Augmentation VS Class Weights

At the end of these two approaches, we obtained two different learning sets:

1. A first learning set composed by:
   (a) Training set balanced with Data Augmentation
   (b) Original Validation set
   (c) Original Test set
2. As second learning set composed by:
   (a) Training set unbalanced but with Class-Weights
   (b) Original Validation set
   (c) Original Test set

To decide which one between Data Augmentation and Class Weight is best for addressing the issue of data unbalancing, we tried to train some pre-trained networks with the two different train splits and compared the results. It turned out that the Data Augmentation approach did not improve the performances, so that we decided to perform all the experiments with the class weights approach. So we kept going with further experiments using that Learning Set.

Chapter 3

CNN from Scratch

We initially attempted to create our own custom neural network architecture from scratch to solve the problem at hand. There are many options for designing a neural network and many hyperparameters to consider. We used a trial-and-error approach to find the best architecture, and then performed further experiments to improve its performance.

3.1 CNN Base Model

We began with a small model that did not perform well and gradually increased its size until it demonstrated good generalization. We will only present the best base architecture, taking into account some of the best experiments that were conducted on this base.
Regarding the input image size, we chose to use 224x224 pixels because the images in the dataset were way too big to handle. We selected a batch size of 32, which is the number of samples processed in each iteration.

After the input layer, we performed a normalization with a Rescaling Layer, so that each pixel value was in the range [0,1]. Then we decided to start with 2 convolutional layers, using the zero-padding technique, in order to give the same importance to each pixel of the image; in fact, even the borders are relevant having cropped the images as much as possible. For the stride we decided to use a value of (2,2). We used an initial learning rate of 0.0001. As activation functions we used ReLU:

$$Relu(z) = max(0, z)$$

As the size of the local receptive fields, we decided to use the default value 3×3. In the first convolutional layer 32 filters are used, and for the consequent convolutional layer the number of filters doubles. Max-pooling is applied after each convolutional level, so that small regions are summarized with a single value. Our architecture uses increasing size for max pooling, with the size of the pooling increasing in the final levels (2×2 in the first layer and 3×3 in the second). This allowed us to decrease the number of network parameters while minimizing accuracy loss. Indeed, we could also increase the stride or avoid zero-padding, but this solution is the one that achieves better performance.

The only thing that does not change is the final layer of the network, that is a dense layer

with a 11 neurons, which utilizes the softmax function for multi-class classification. Throughout our experiments we used dropout as a regularization technique, as it was found to be the most effective for our problem in various cases.

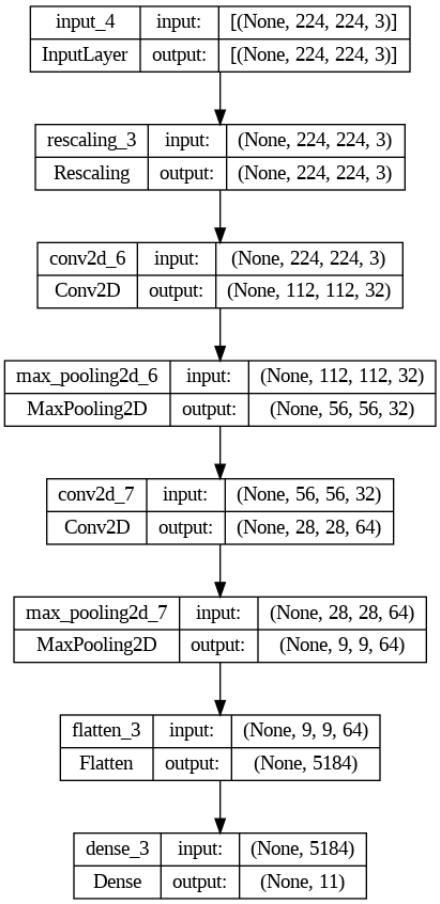


Figure 3.1: Base CNN Structure

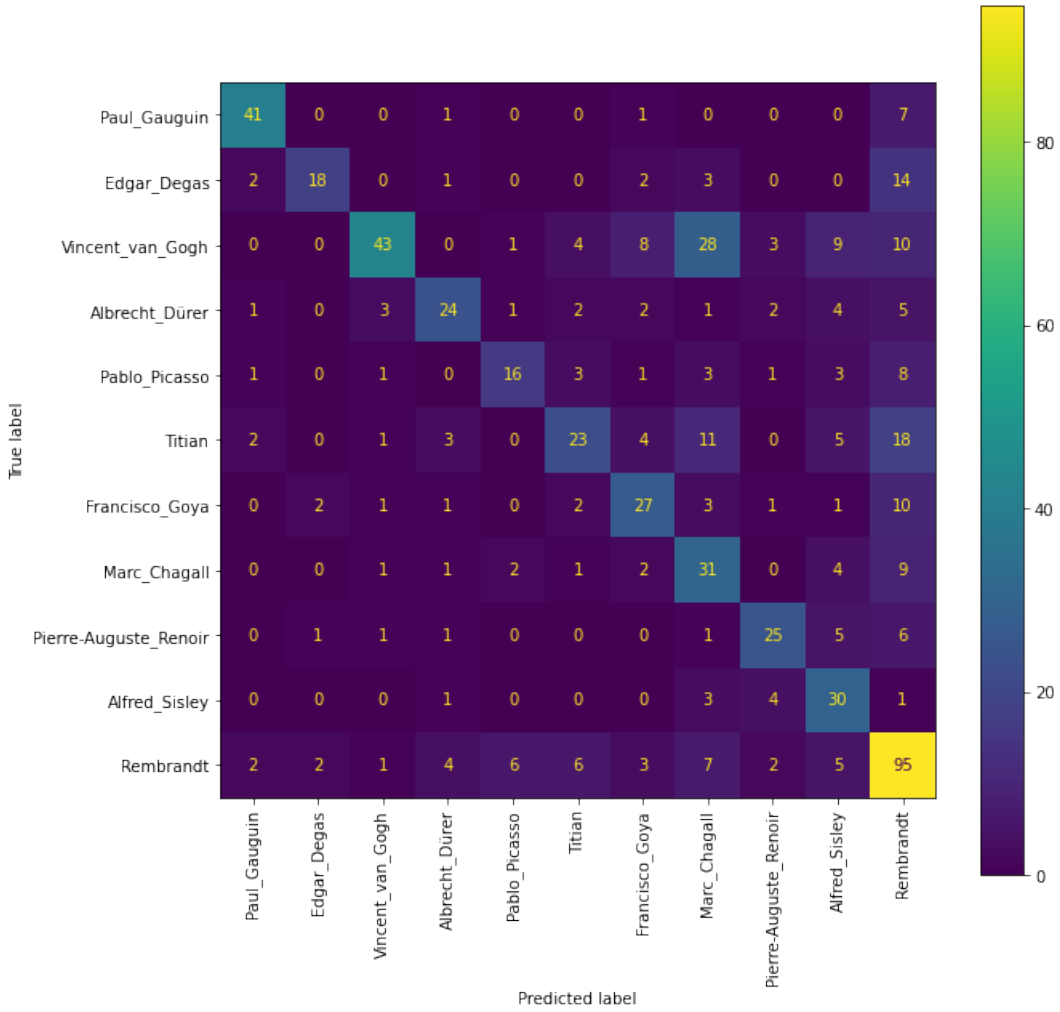
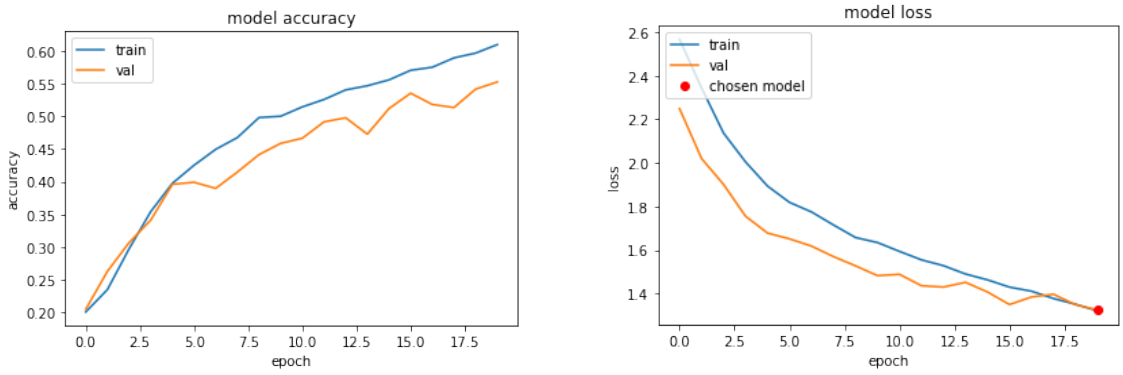


Figure 3.2: Base CNN Confusion Matrix



(a) Model Accuracy

(b) Model Loss

Figure 3.3: Model History

We can spot that there are two classes that suffer from missclassification that are Marc

Chagall and Rembrandt. Chagall is often missclassified as Van Gogh, this could be caused by the similarity in the style and the used colors. While Rembrandt is often missclassified as Titian, their art is mostly composed by human portraits and for the network is not easy to spot the differences.

The following tables refers to the best model saved during the training, that is depicted with a red point in the Model Loss graph.

Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Test Accuracy
1.32	0.61	1.32	0.55	0.56

As we can see from the plot on the left the model model suffers from under-fitting since the last training accuracy barely reached 60%. We could have trained the model for more epochs but we preferred to increase the learning rate to speedup the training process.

3.2 Base CNN with higher LR

We kept the same structure as before, we only increased the learning rate from 0.0001 to 0.001 in order to avoid the model to converge too slowly. We can see that the model starts to learn way faster than before:

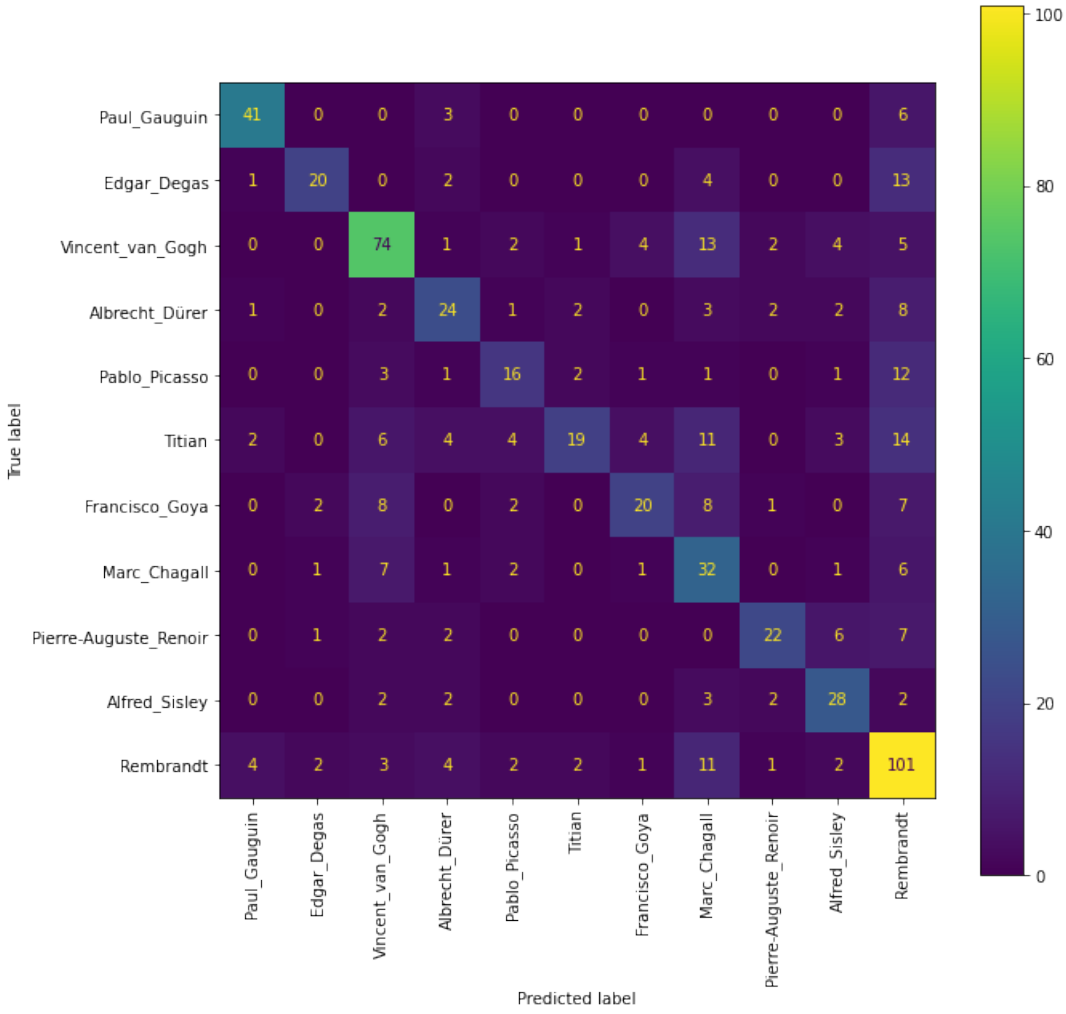
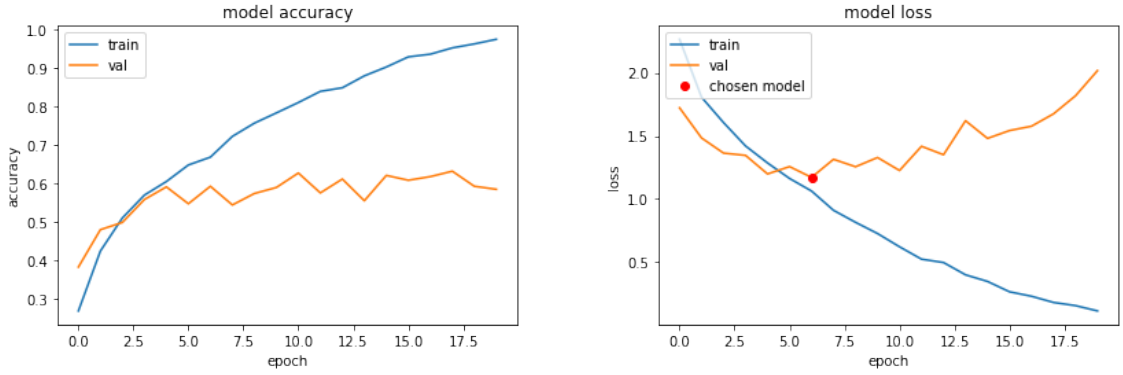


Figure 3.4: Base CNN Confusion Matrix with higher LR

We have a slight improvement in the two aforementioned classes, but there is still some noise.



(a) Model Accuracy

(b) Model Loss

Figure 3.5: Model History

As expected, we obtained with the same number of epochs an higher train accuracy value. Anyway, now we have an strong gap between train and validation accuracy and in the model loss graph we can notice that the model starts to learn the noise very early (third epoch). This is a clear sign that this model suffers from overfitting. To tackle this problem we could try to reduce the size of the features, add regularizers, try data augmentation or introduce the early stopping.

Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Test Accuracy
1.06	0.67	1.17	0.59	0.61

Regarding the results, we obtained slightly better performances, especially on test set (from 0.56 to 0.61).

For trying to fix the overfitting we decided to adopt the early stopping mechanism and adding a dropout layer, while to increase the validation accuracy we inserted an extra Convolutional Layer with 128 neurons.

3.3 Dropout, Early Stopping, 3 CL

As mentioned before, we added an extra convolutional layer and adopted the early stopping (patience = 5) and dropout, resulting in the following model structure:

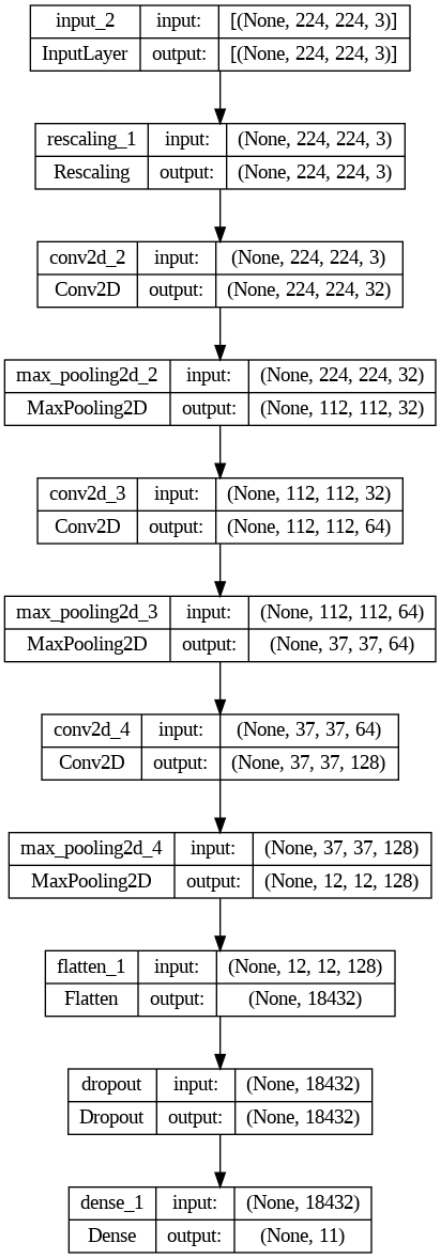


Figure 3.6: CNN Conv=3, Early Stopping and Dropout Structure

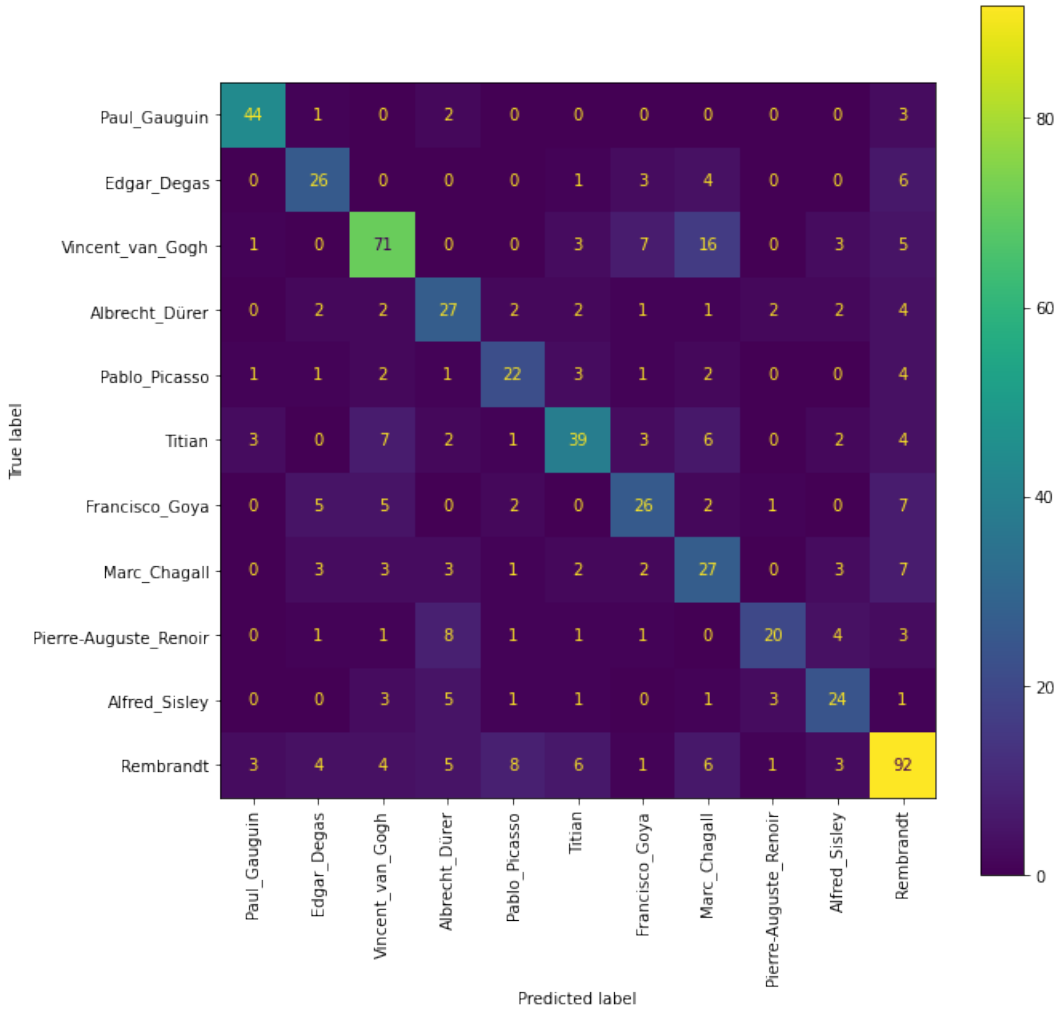
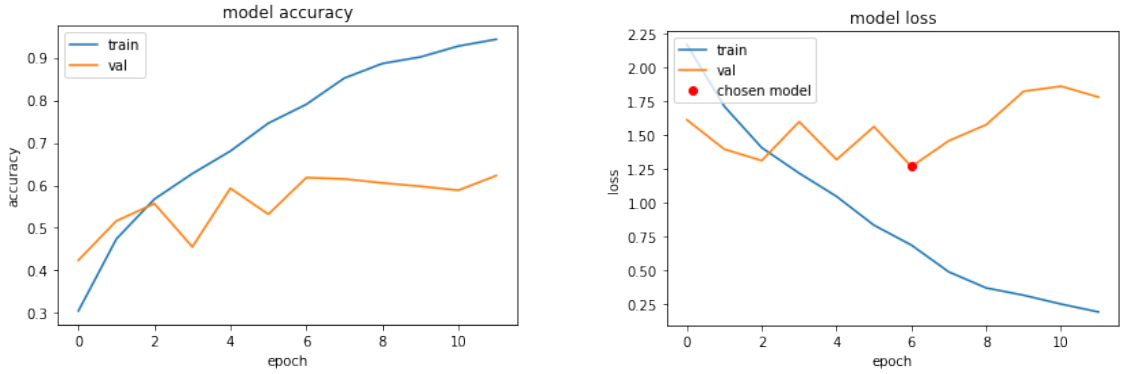


Figure 3.7: CNN Conv=3, Early Stopping and Dropout Confusion Matrix

As we can see from the confusion matrix, the network is now performing significantly better on several classes, lowering the missclassification rate.



(a) Model Accuracy

(b) Model Loss

Figure 3.8: Model History

Looking at the first plot, we notice that the model still suffers from overfitting as denoted by the presence of the gap between train accuracy and validation accuracy, even if it has been slightly reduced from the previous trial. In addition, this time the training stopped early, at epoch 12 instead of 20, avoiding the model to learn noise. The main drawback of this setup is the model loss for the validation, that remains higher (1.26) with respect to the training error (0.68).

Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Test Accuracy
0.68	0.79	1.26	0.61	0.63

We obtained an increased value of accuracy both for training (+0.12) and validation (+0.02). Anyway, we still need to try to raise the validation accuracy values, so now we try to add a Dense Layer of 128 neurons to increase the learning capability of the network, as we know that Dense layers add an interesting non-linearity property.

3.4 Dropout, Dense, Different Strides

As mentioned above, this time we tried to deepen the network but keeping the total number of parameters low. So, we added a Dense layer but we changed the strides values and max poolings' kernels:

- **First Conv2D from stride (1,1) to (7,7)**
- **First MaxPooling layer from stride (1,1) to (4,4)**

Applying these modifications resulted in lowering the total number of parameters **from 296,011 to 160,331**.
The intuition is applying bigger strides it will downsamples the image, lowering the number of parameters.

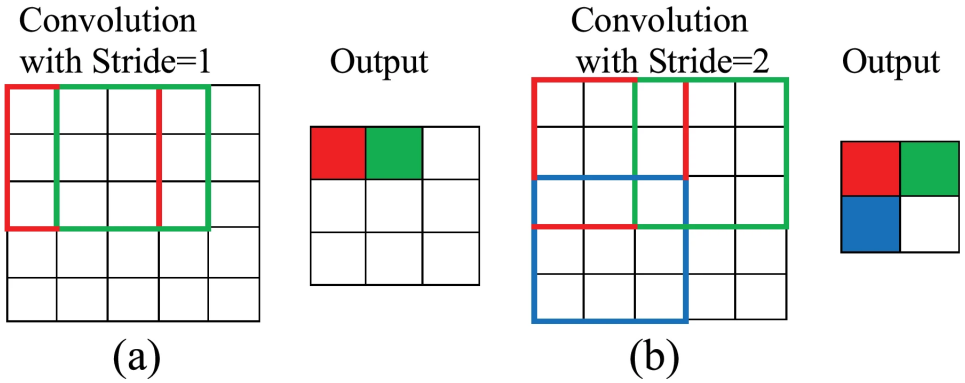


Figure 3.9: Intuition behind increasing the stride in the first ConvLayer, it will result in a lowered number of total parameters

Putting them in this first layer results in applying convolution and downsampling at the same time, so that the operation becomes significantly cheaper computationally.
The overall structure becomes:

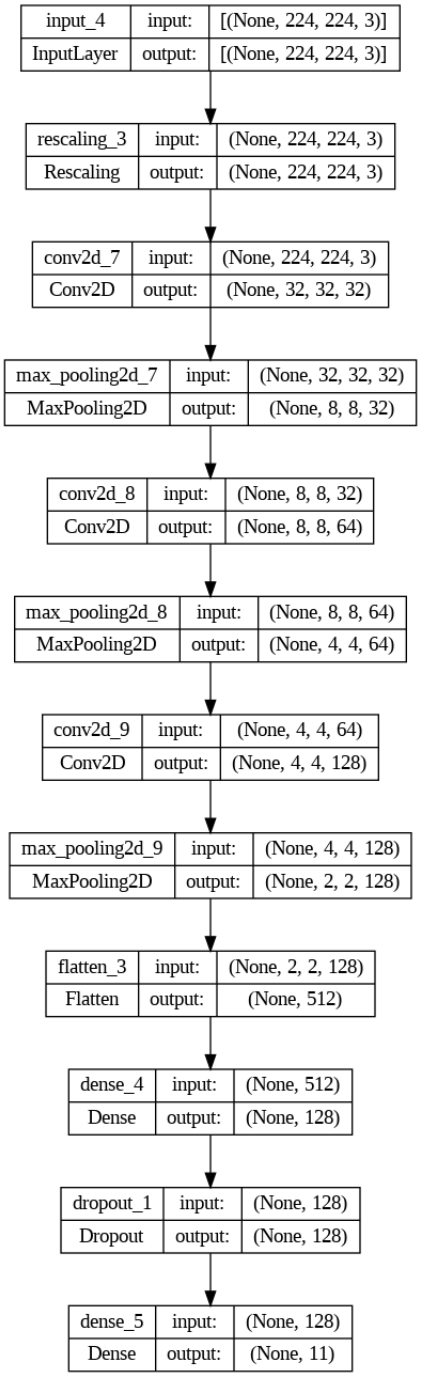


Figure 3.10: CNN Conv=3, Dropout and Dense Structure

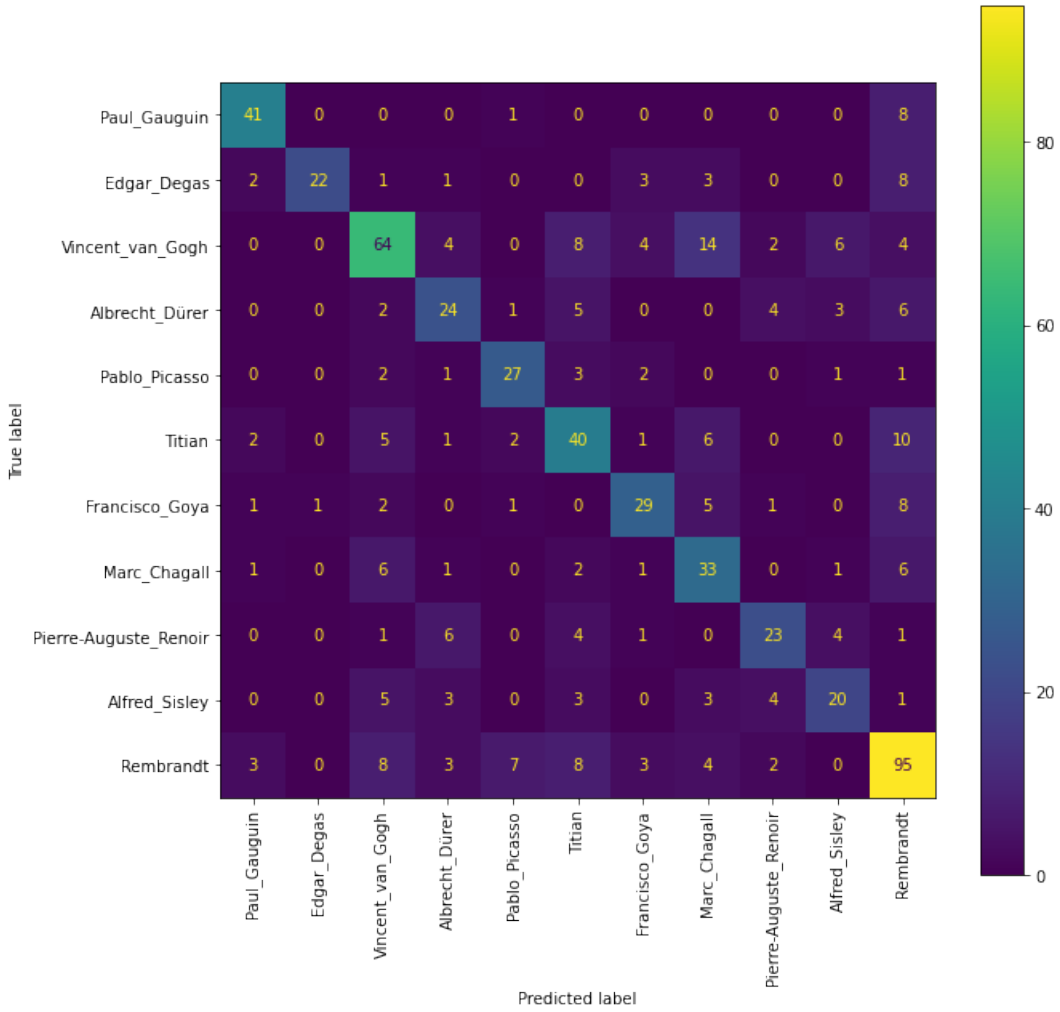
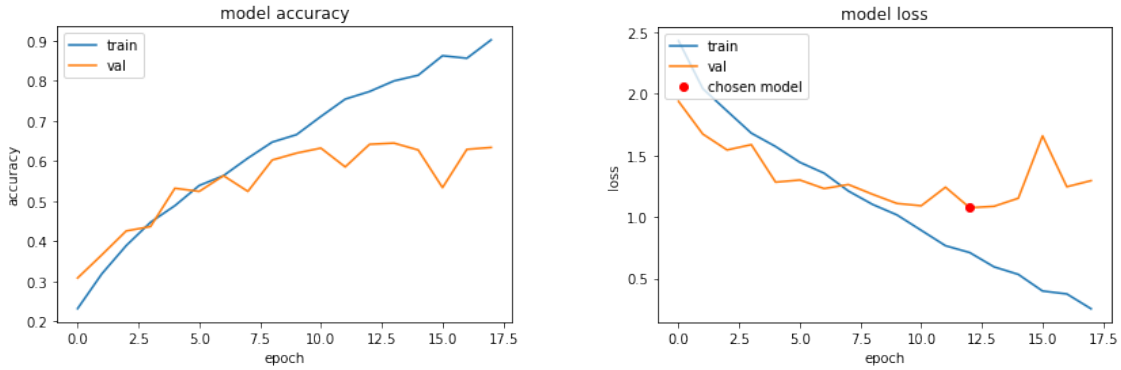


Figure 3.11: CNN Conv=3, Dropout and Dense Confusion Matrix

From the matrix if difficult to tell whether we obtained a better model or not, so let's analyze the other metrics:



(a) Model Accuracy

(b) Model Loss

Figure 3.12: Model History

We lowered even more the gap between training and validation accuracies, even if the latter is still quite low. While in the plot on the right we obtained a better error trend (from 1.26 to 1.07) and the model starts to overfit later (at epoch 10) with respect to the previous experiment (at epoch 6).

Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Test Accuracy
0.71	0.77	1.07	0.64	0.64

Apparently it seems that the model does not improved a lot, but actually we succeeded in lowering the complexity of the model (less parameters) while obtaining better results (+0.01 in test accuracy, +0.03 validation accuracy and -0.19 in validation loss).

3.5 Regularization

In order to reduce the zig-zag behaviour of the validation accuracy trend we decided to try to apply the **L2 regularization** to all the convolutional layers. The L2 regularization technique, also known as Ridge regression, adds "squared magnitude" of coefficient as penalty term to the loss function:

$$Loss = Error(Y - \widehat{Y}) + \lambda \sum_{1}^{n} w_i^2$$

We maintained the exact same structure as before, and these are the results that we obtained.

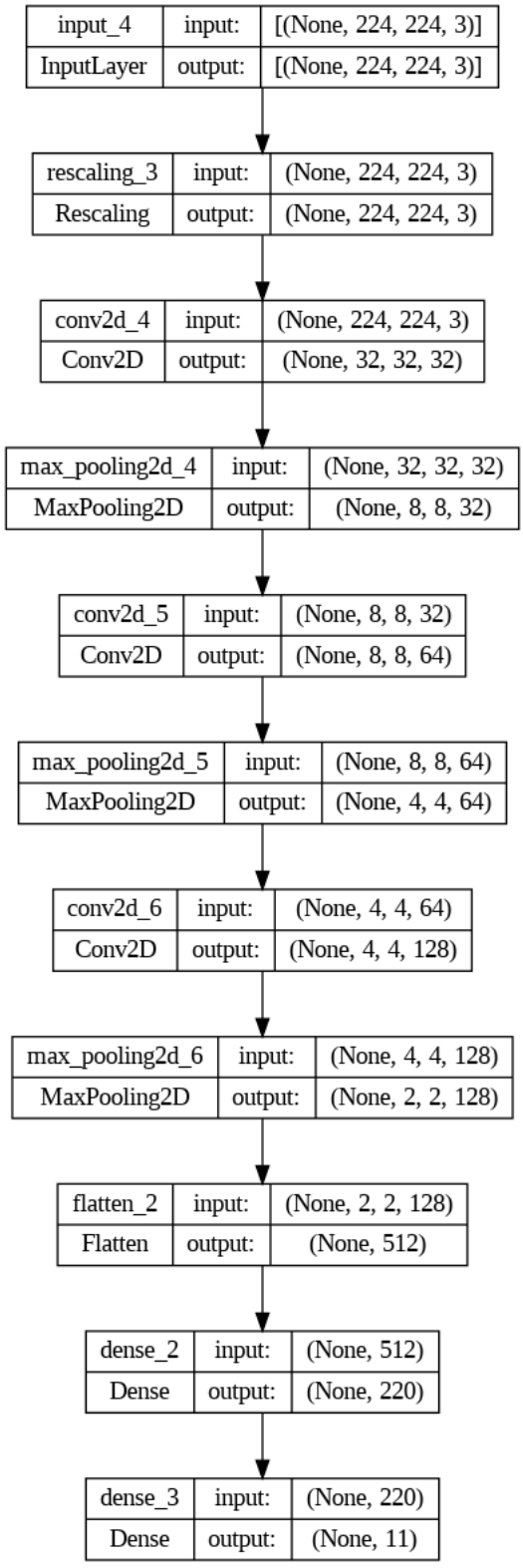


Figure 3.13: Structure of the CNN with L2 Regularization on all the ConvLayers

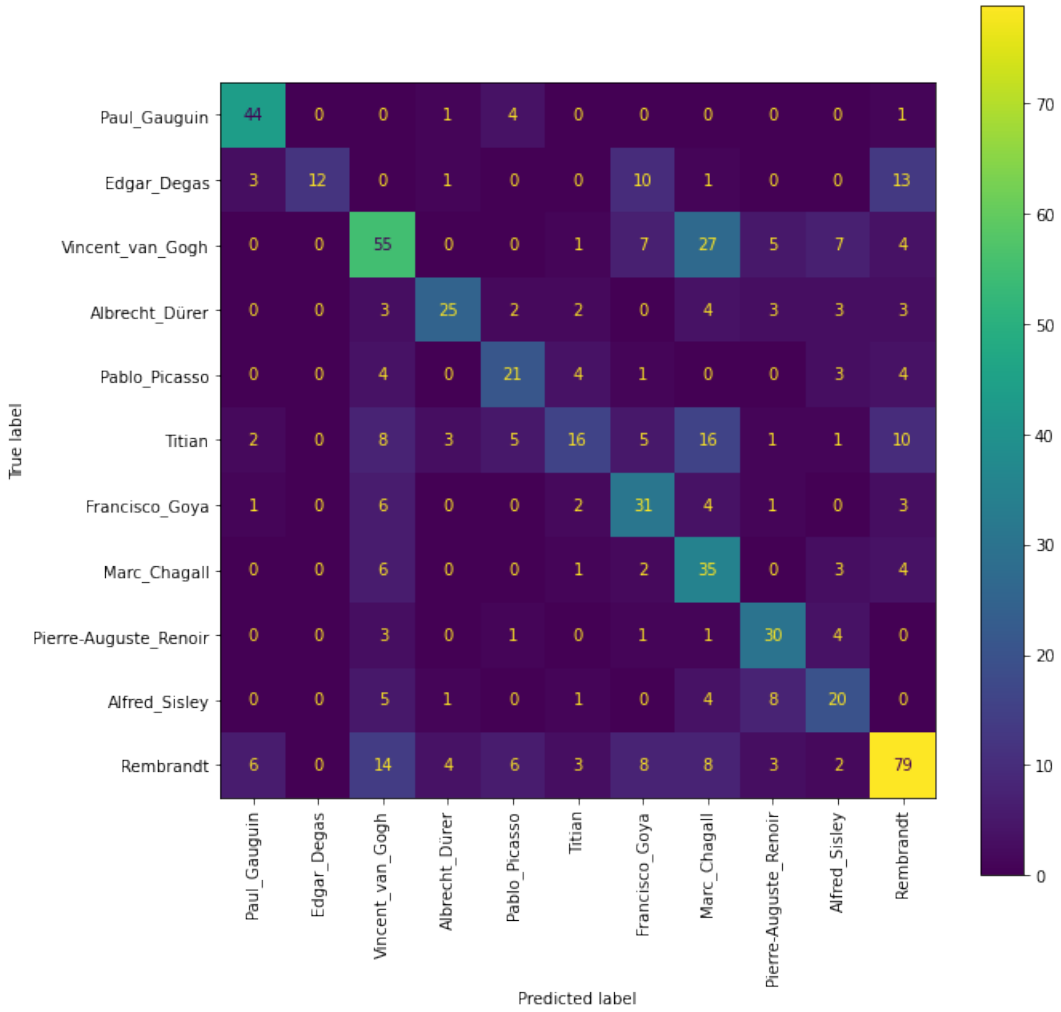
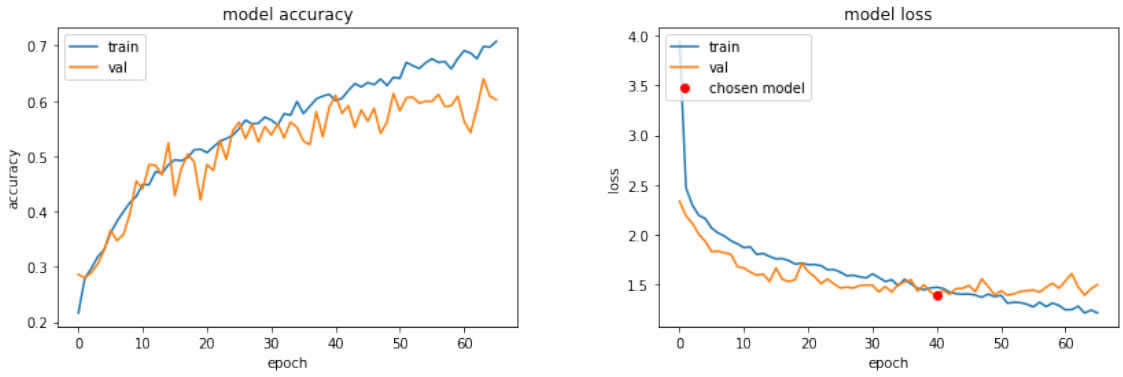


Figure 3.14: CNN with L2 Regularization on all the ConvLayers, Confusion Matrix

Looking at the confusion matrix, we can see that the Regularization introduces noise during training, that end up with many more missclassified labels.



(a) Model Accuracy

(b) Model Loss

Figure 3.15: Model History

The plots' trend is consistently better regarding the gap between train/ validation accuracy and train/validation errors, fixing the missing convergency issue; anyway, we can observe that the model suffers again from underfitting as the train accuracy barely reaches the 70%. We also needed to increase significantly the number of epochs because of the slow convergence (caused by L2 reg) even with an high lr (0.001). Despite this, we were not able to achieve a acceptable accuracy on the validation set, that remains stuck at $\sim 60\%$.

Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Test Accuracy
1.47	0.60	1.39	0.61	0.56

Those are the best results we could obtain by running several trials; specifically, w.r.t. the previous model we deleted the dropout layer and increased the number of hidden neurons from 128 to 210.
From the table we can observe that the network underperforms the previous experiment in all the metrics: train loss (+0.76), train accuracy (-0.17), test accuracy (-0.08), validation loss (+0.32) and validation accuracy (-0.03).

3.6 Dropout, Deeper Dense

As we saw in the previous section, adding regularizers does not help in improving performances. We took a step back and returned to model in section 3.4, we decided to deepen the dense layer from 128 to 190 (increasing the parameter from 160,331 to 192,819) following the same idea adopted in section 3.4. We also set the dropout rate to 0.5 as suggested by the inventors of this technique in their paper [8].

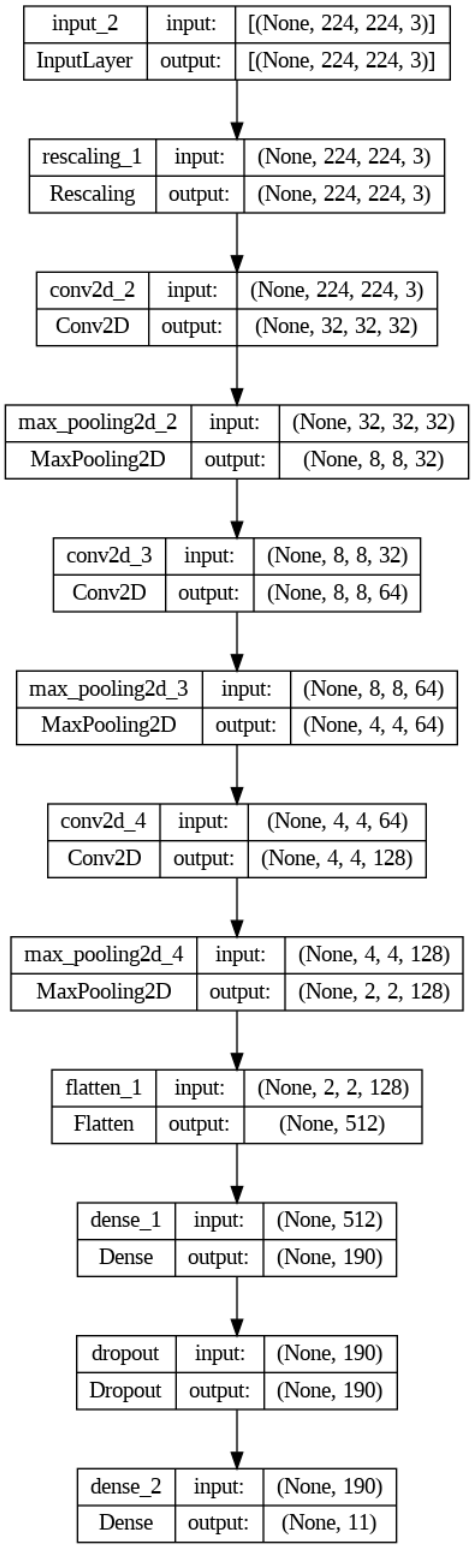


Figure 3.16: CNN Conv=3, Dropout and Deeper Dense Structure

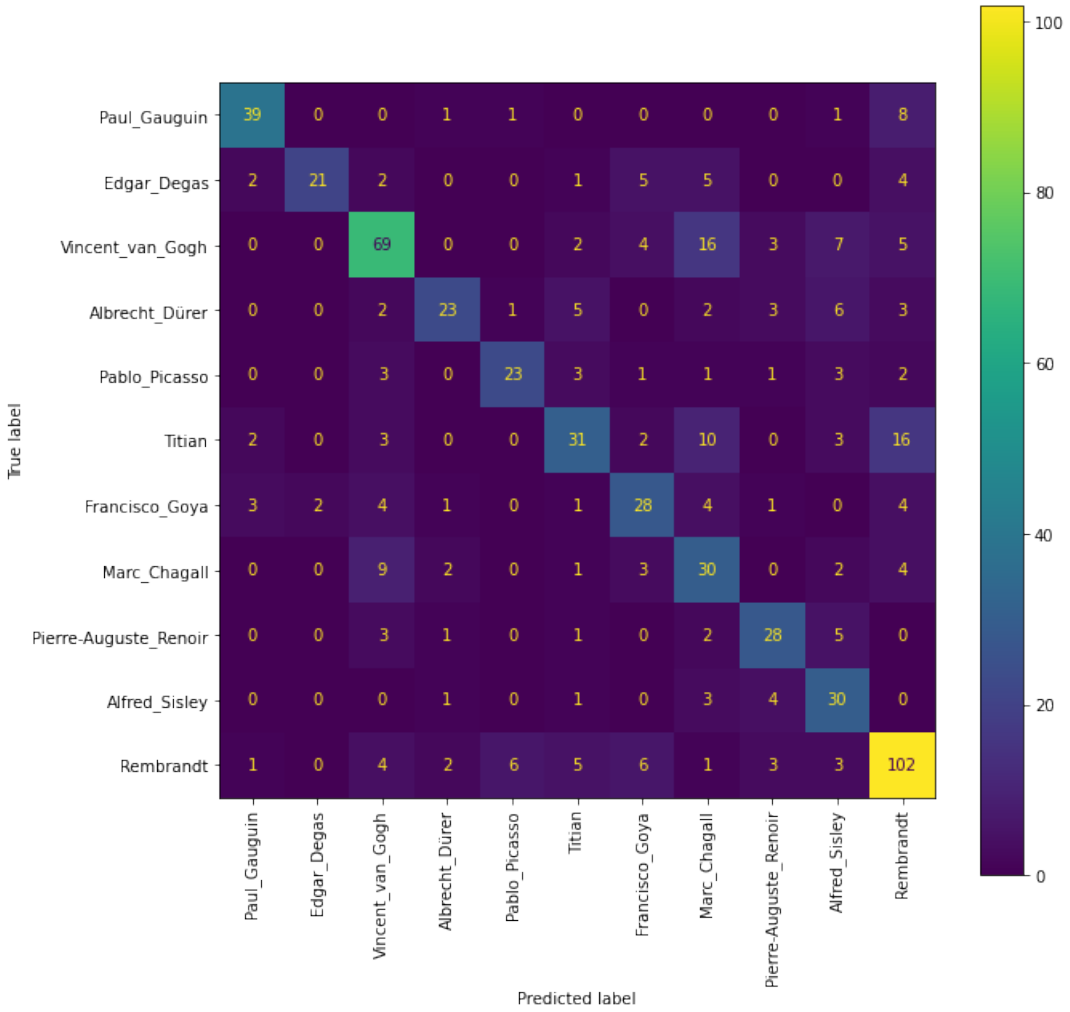
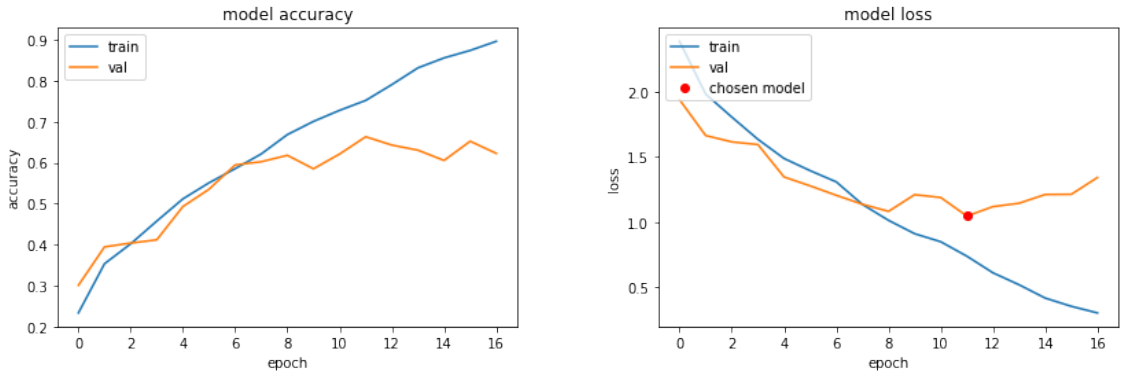


Figure 3.17: CNN Conv=3, Dropout and Deeper Dense Confusion Matrix



(a) Model Accuracy

(b) Model Loss

Figure 3.18: Model History

Once again we were able to reduce the gap between the two accuracies. Instead, we did

not obtained a better trend for the validation loss with respect to figure 3.12.

Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Test Accuracy
0.73	0.75	1.04	0.66	0.65

We managed to obtain the best performances so far, specifically: validation loss (-0.03), validation accuracy (+0.02) and test accuracy (+0.01).

3.7 2x Dense Layer

We tried to split the single Dense layer into to different ones followed by two Dropout layers. Since the excluded neurons with dropout changes in each epochs (picked randomly by percentage), here the idea is to introduce an additional layer of dropout after a new Dense one, in order to let the network have more combination of different neurons throughout the training phase. In fact, in each epoch, we will have different nodes selected by the 2 dropout layers in each Dense layers.

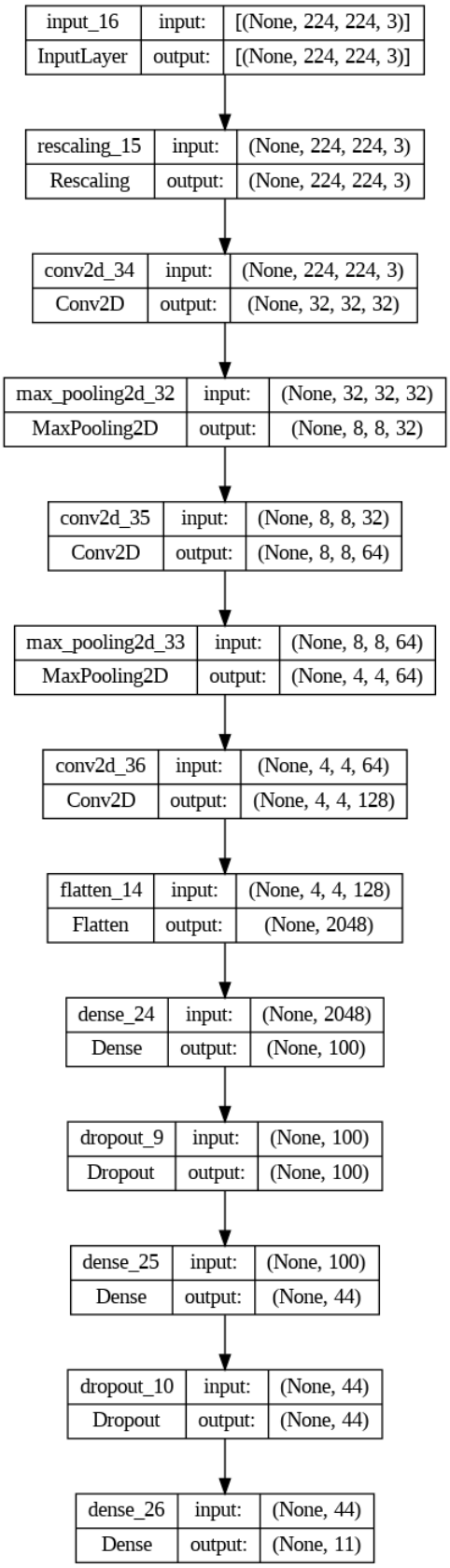


Figure 3.19: CNN Conv=3, 2x Dense and 2x Dropout Structure

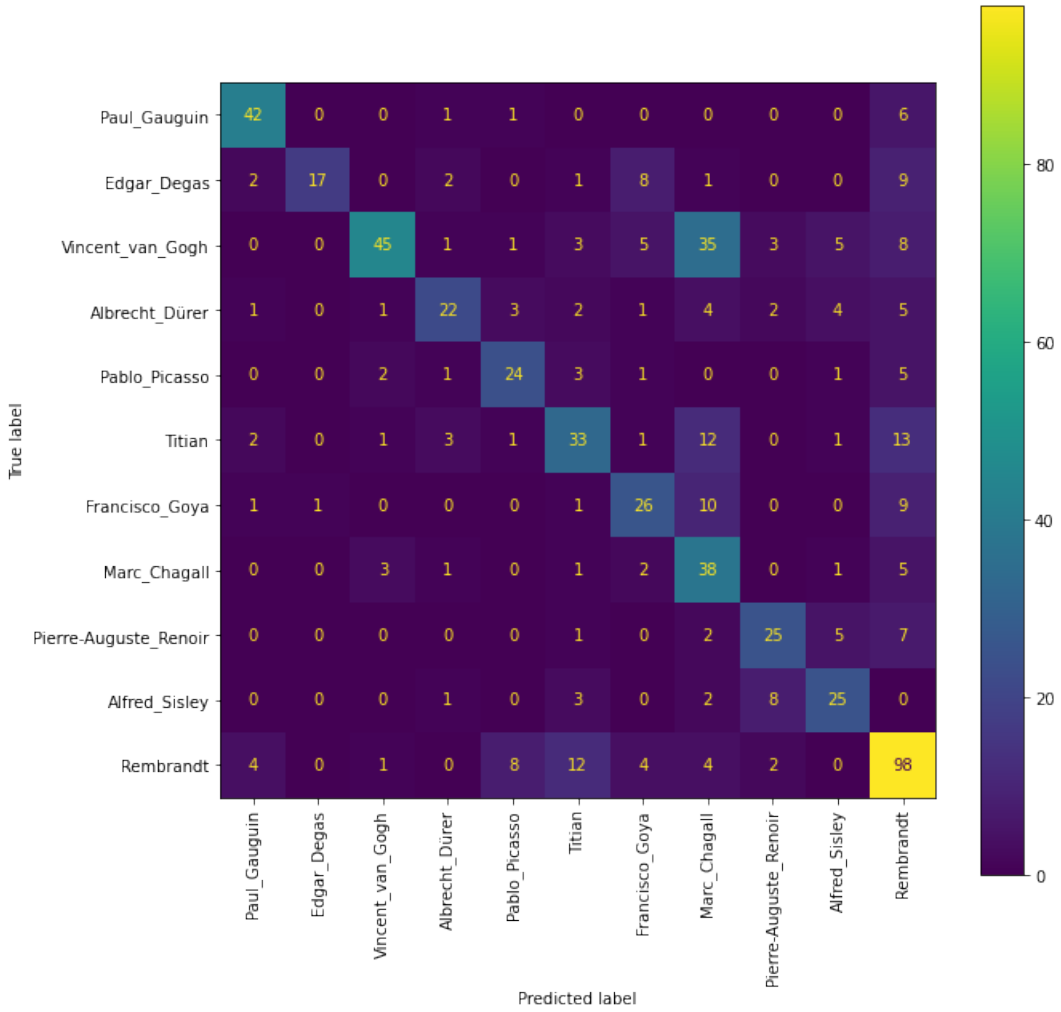
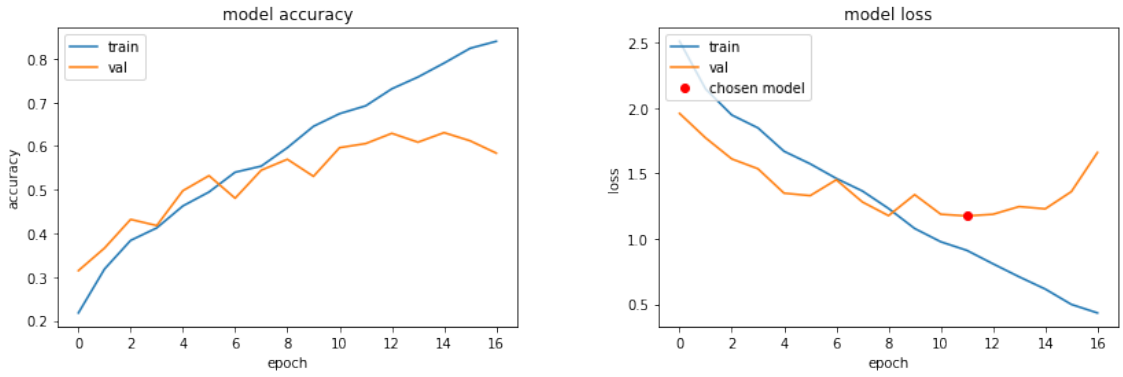


Figure 3.20: CNN Conv=3, 2x Dense and 2x Dropout Confusion Matrix



(a) Model Accuracy

(b) Model Loss

Figure 3.21: Model History

From the plot on the left we can see that the validation accuracy trend is now way more

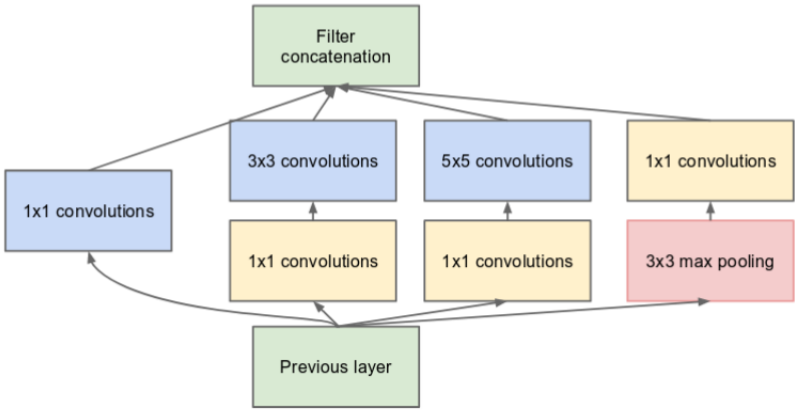
flat, achieving lower values than before. Regarding the model loss, we can observe that the validation error starts a steeper increase in learning noise (U shape) w.r.t. of the previous models.

Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Test Accuracy
0.91	0.69	1.17	0.60	0.60

The performances with the additional Dense layer decreased significantly, probably due to the fact that we increased the parameters from 192,819 to 303,087 and those are too much considered that is about 100 times the number of our training images (approx. 3000). Thus, we discard this experiment.

3.8 Inception Layers

Inception modules are unique convolutional blocks that consist of multiple layers with different kernel sizes and a max pooling, rather than just one layer. The idea behind this design is to allow the network to learn which filter to use at each layer, rather than relying on a predefined hyper-parameter, which can negatively impact performance. Google first introduced inception modules in their 2014 GoogleNet [9], and further refined them in Inceptionv3 and Inceptionv4. The aim of this section is to design and train a Convolutional Neural Network with Inception Layers and evaluate its accuracy.



(b) Inception module with dimension reductions

Figure 3.22: Single Inception Layer Structure

3.8.1 Single Inception

In this paragraph, we construct a model starting from the section 3.6, our best model so far, but incorporating custom-designed inception modules. At each module, the network has the ability to choose between a 3x3 or 5x5 convolution or a 3x3 max-pooling with a stride of 1.
In our Inception architecture we utilized 1x1 convolutional filters for reducing the dimensionality in the filter dimension. These 1x1 conv layers can either increase or decrease the filter space dimensionality, and in the Inception architecture, they effectively reduce the filter dimension space, not the spatial dimension space. A 1x1 convolution maps an input pixel and its channels to an output pixel without considering surrounding pixels. It's frequently utilized for lowering the depth channel count since large depth values slow down multiplication. To better explain the intuition of 1x1 ConvLayer we applied, we attach the following image:

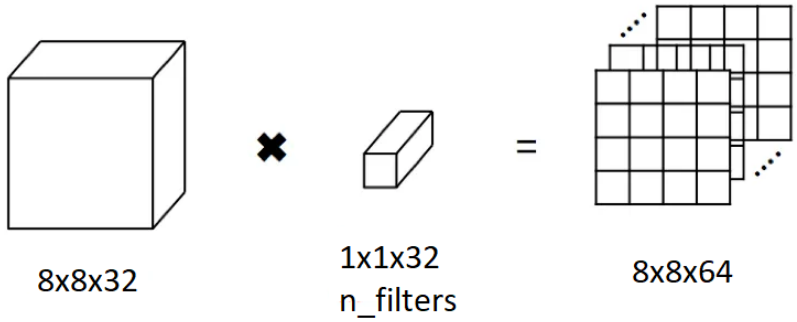


Figure 3.23: 1x1 ConvLayer helps deepen the network without affecting the structure

The resulting architecture is as follows:

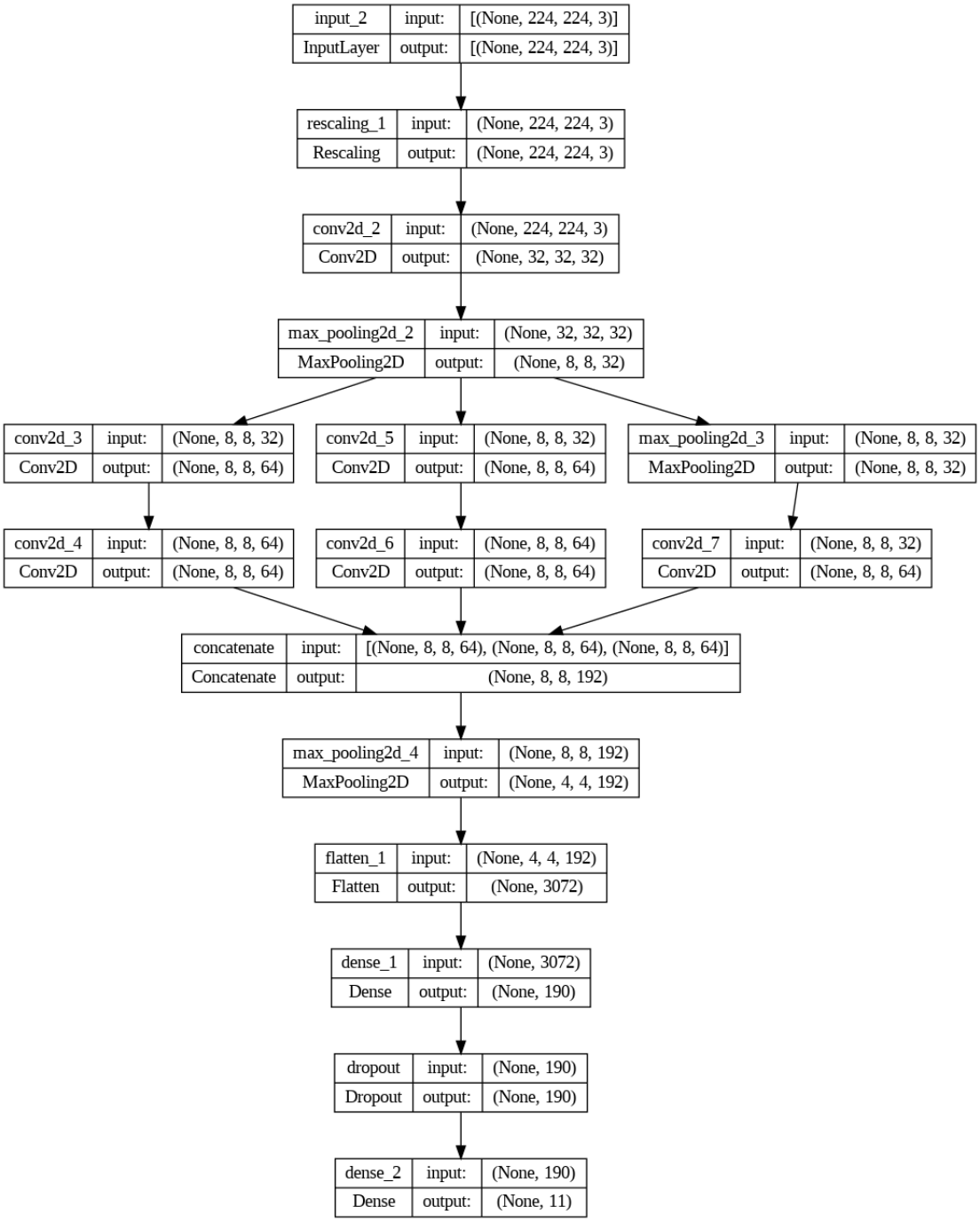


Figure 3.24: Single Inception Layer Structure

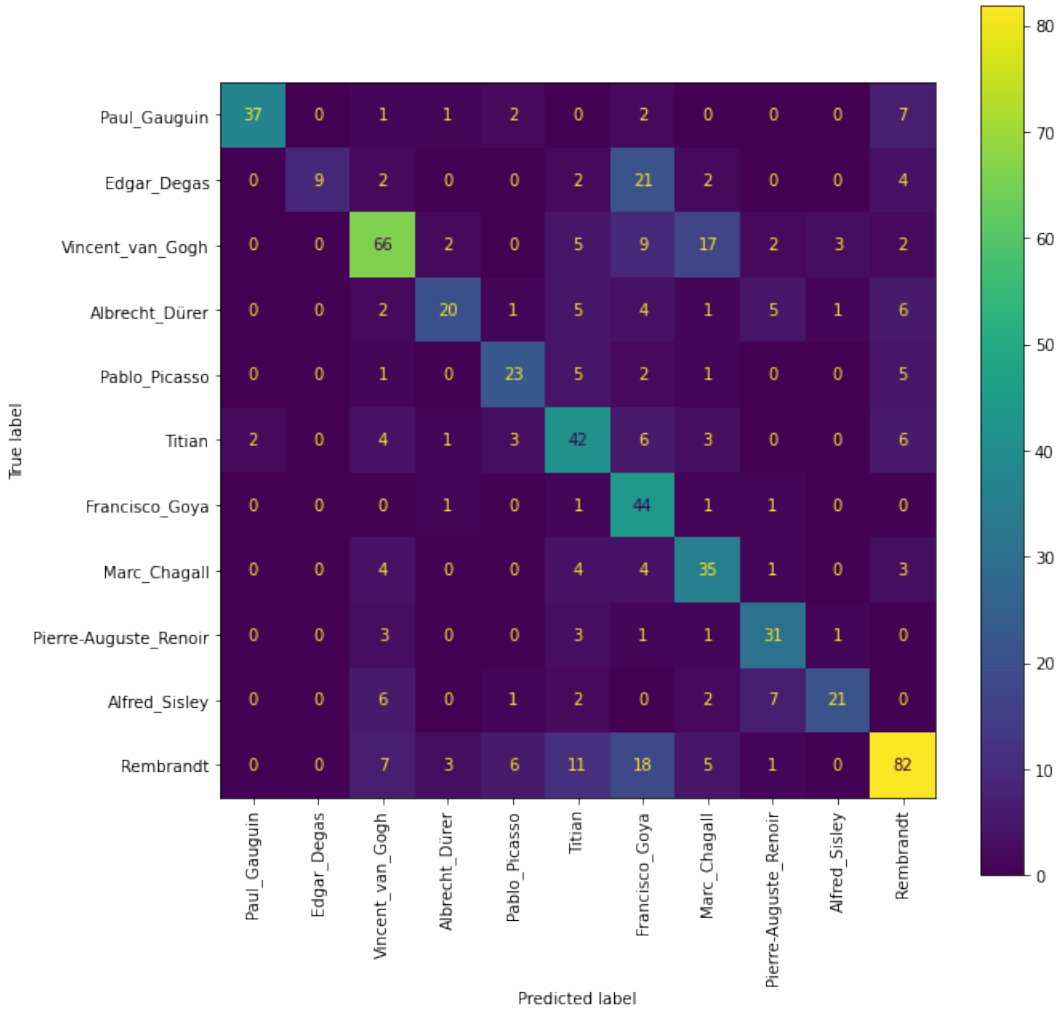
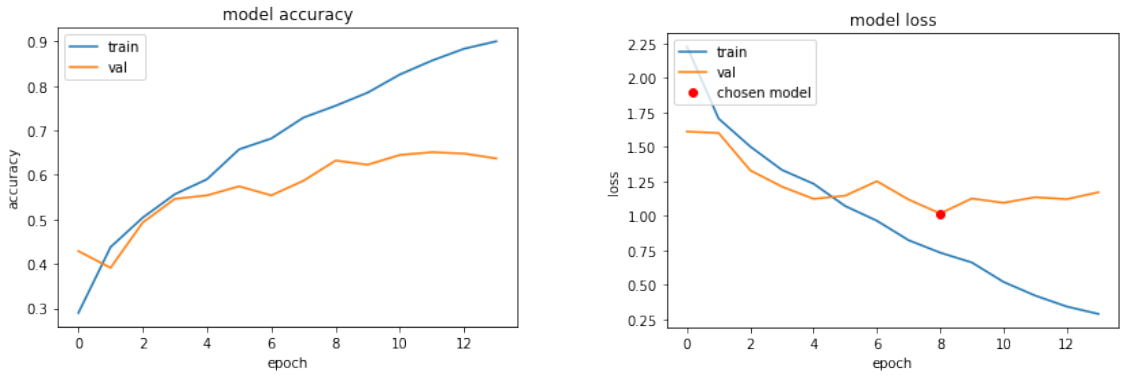


Figure 3.25: CNN with Single Inception Layers, Confusion Matrix



(a) Model Accuracy

(b) Model Loss

Figure 3.26: Model History

From the plots we cannot observe any kind of improvements, with the missing convergence

issue still present between the two accuracy trends.

Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Test Accuracy
0.73	0.75	1.01	0.63	0.62

Coherently with the plots above, we observe a general decrease in performance w.r.t. our best model without inception layers.

3.8.2 Double Inception

In this paragraph, we construct a model similar to the previous model but we added an extra inception layer with a maxpooling layer between the two inceptions.
The architecture is as follows:

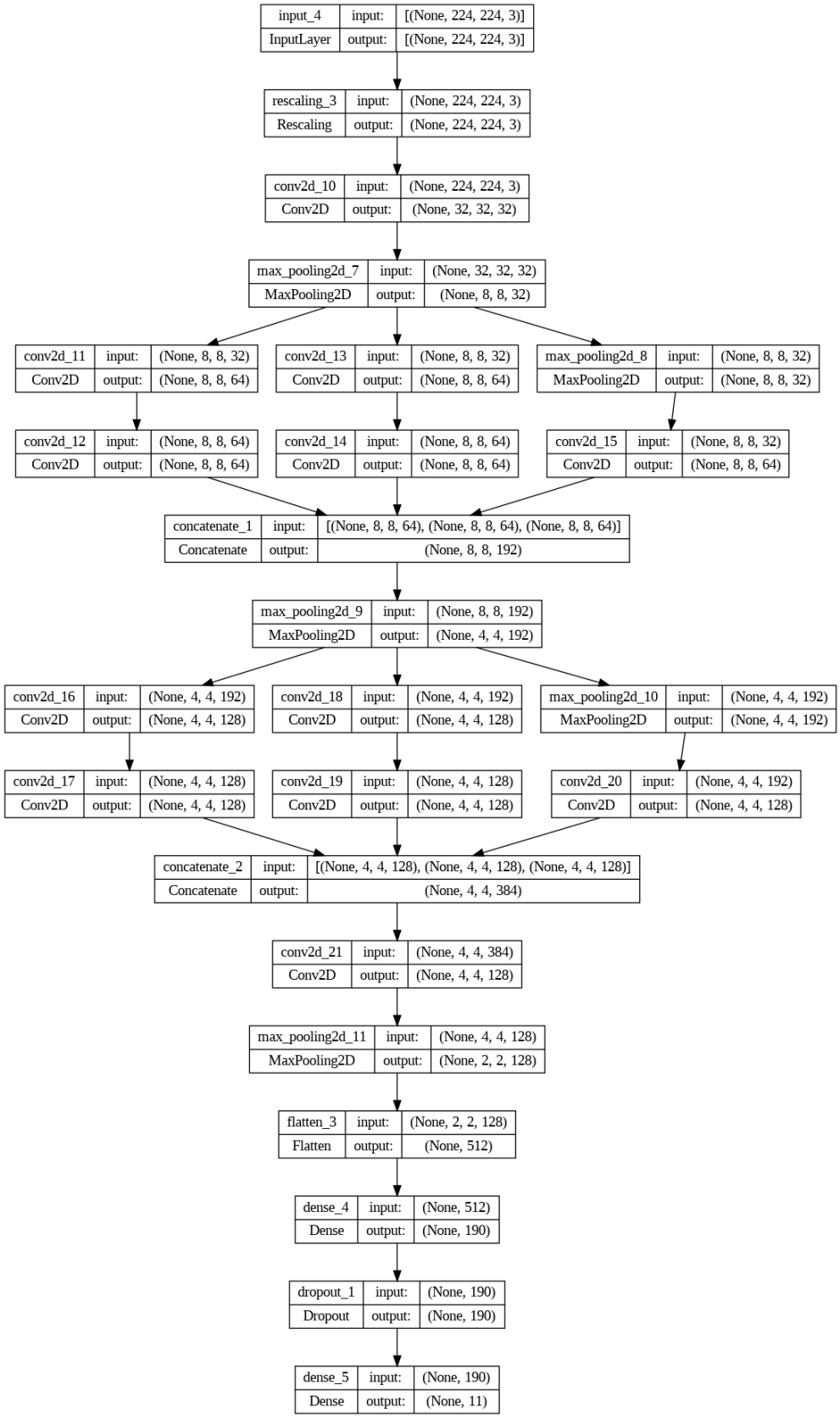


Figure 3.27: Double Inception Layers Structure

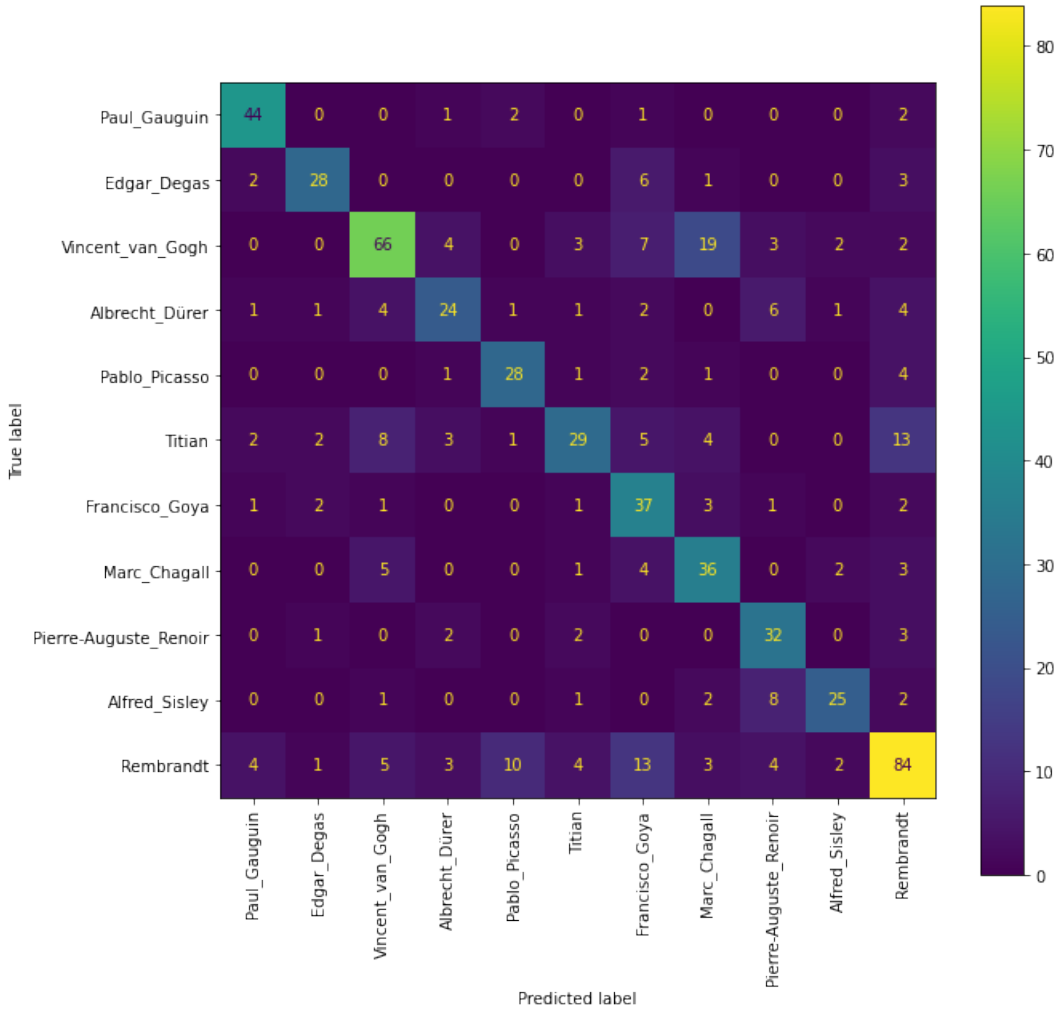
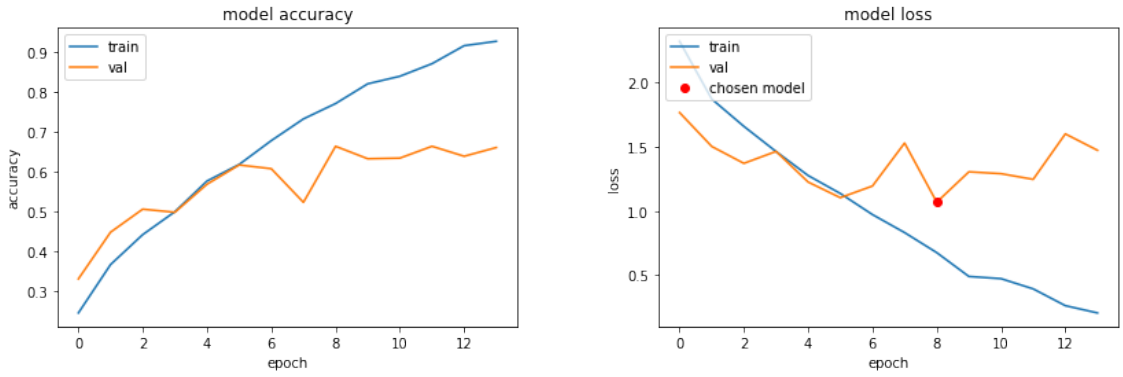


Figure 3.28: CNN with Double Inception Layers, Confusion Matrix



(a) Model Accuracy

(b) Model Loss

Figure 3.29: Model History

The gap is now lower than before, and the validation accuracy values are approaching

70%. Regarding the model loss, does not seem to change w.r.t. the previous experiment, with an error loss that is not able to fall below the 1.00 wall.

Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Test Accuracy
0.68	0.77	1.07	0.66	0.66

The network performs very similarly to our best model, but we managed to obtain a better accuracy on the test set (+0.01). We can conclude that this model resembles our best architecture so far.

Chapter 4

Hyperparameter Optimization

We decided to explore the potential of our best models and see if we could get the most out of them through Hyperparameter Tuning (process of selecting the right set of parameters for our machine learning model).

We also wanted to see if the results we got from our implementation of the from scratch CNN was comparable with some ad-hoc optimized models. So, we tried running the hyper-parameter optimization algorithm on the best model we were able to build. We had a few options to choose from, such as Grid Search, Hyperband, Bayesian Model and Random Search. We exploited the **Keras Tuner** library, that helped us pick the optimal set of hyperparameters. With the configuration of our model (cap. 3.4) we run Random Search on it, aiming at optimizing the following hyper-parameters:

- **n neurons**: Number of neurons in the FC layer, chosen as an integer value between 0 and 256 with a step size of 16, immediately following flattening and before classification.
- **Dropout rate**: In this layer, the activation function is a float value between 0 and 0.6 that represents the dropout rate.
- **Activation function**: we previously only considered Rectified Linear Units (ReLU, but now we try to choose from either the Exploratory Linear Unit (ELU) or the Generalized Linear Unit (GELU).
- **lr**: The optimizer's learning rate. Previously, we always utilized ADAM's default value of 1e-3; going forward, the tuner will select a float value between 5e-5 and 5e-3.

After running the tuner, we got the 10 best models and we visualized the best combinations of hyper-parameters of the top-3, which are reported below:

```
def buildmodel(hp):

  # set the hyperparameters
  activationhp = hp.Choice('activationfunction', values=['relu',
  ↪  'elu', 'gelu'])
  nneuronshp = hp.Int('FClayer', minvalue=0, maxvalue=256, step=16)
  dropoutratehp = hp.Float('dropoutrate', minvalue = 0, maxvalue =
  ↪  0.5)
  lrhp = hp.Float('learningrate', minvalue=5e-5, maxvalue=5e-3)

  #definition of the parametric model
  inputs = keras.Input(shape=(224, 224, 3))
  x = layers.Rescaling(1. / 255)(inputs)
  x = layers.Conv2D(filters=32, kernelsize=(3, 3), strides=(7,7),
  ↪  padding="same", activation=activationhp)(x)
  x = layers.MaxPooling2D(poolsize=(4, 4), strides=4)(x)
  x = layers.Conv2D(filters=64, kernelsize=(3, 3), padding="same",
  ↪  activation=activationhp)(x)
  x = layers.MaxPooling2D(poolsize=(2, 2))(x)
  x = layers.Conv2D(filters=128, kernelsize=(3, 3), padding="same",
  ↪  activation=activationhp)(x)
  x = layers.MaxPooling2D(poolsize=(2, 2))(x)
  x = layers.Flatten()(x)
  x = layers.Dense(nneuronshp)(x)
  x = layers.Dropout(dropoutratehp)(x)
  outputs = layers.Dense(len(CLASSES), activation='softmax',
  ↪  name='predictions')(x)
  model = tf.keras.Model(inputs=inputs, outputs=outputs)

  ↪  model.compile(optimizer=tf.keras.optimizers.Adam(learningrate=lrhp),
  ↪  loss='categoricalcrossentropy', metrics=['accuracy'])
  return model
```

In table 4.1 the best three hyperparameter sets are reported.

Best Models	Activation Function	FC Layer	Dropout Rate	Learning Rate	Val Score
first	relu	112	0.28	0.001	0.653
second	relu	80	0.27	0.0007	0.651
third	elu	16	0.27	0.001	0.647

Table 4.1: Top three hyperparameter sets

As we can see, even with 30 different trials with hyperparameter optimization, the best results still oscillate around 0.65 of validation accuracy.
Finally, we evaluated our best model onto the test set, obtaining the following results:

```
# Get the top model
bestmodel = tuner.getbestmodels(nummodels = 1)[0]
bestmodel.evaluate(testimages)

test loss: 1.1612 - test accuracy: 0.6189
```

So, even the results upon test set are comparable with the ones we obtained with our from scratch model (even worse). These values are not very satisfactory, and this brought us to think that the main problem is because of the small size of our dataset and also because of the complex task we are trying to solve (multiclassification problem with 11 classes + very similar paintings style from several artists).

Chapter 5

Pre-trained Models

As our results using a CNN from scratch were not very good, especially when compared to current state of the art, we believe we can improve performance by using transfer learning. Given that we have a relatively small training dataset, it is challenging to create networks that can generalize well. By using pre-trained networks, we aim to make this task easier, by experimenting with both feature extraction and fine-tuning, using a trial-and-error approach for the networks mentioned below.

VGG16 is a convolutional neural network model proposed in 2014 by K. Simonyan and A. Zisserman from the University of Oxford in the paper "Very Deep Convolutional Networks for Large-Scale Image Recognition" [10].

ResNet50, short for Residual Network with 50 layers, is a specific type of neural network that was introduced in 2015 by K. He, X. Zhang, S. Ren and J. Sun in their paper "Deep Residual Learning for Image Recognition" [11].

InceptionV3, an image recognition model that has been shown to attain greater than 78.1% accuracy on the ImageNet dataset. The model is the culmination of many ideas developed by multiple researchers over the years. It is better than its previous versions like the Inception V1 model and other Models like Resnet. [12]

5.1 VGG16

The original architecture of VGG16 is shown if Fig. 3.1. The convolutional base consists of five blocks, each with 2 or 3 convolutional layers using 3x3 filters. These blocks are followed by a densely connected classifier that includes three layers, with the final layer being a 1000-node softmax layer. All hidden layers utilize the rectification (ReLU) activation function. Additionally, between blocks, max-pooling is applied on a 2x2 window, with a stride of 2. The network is designed to take in a fixed input size of 224x224x3 and outputs a vector of 1000 probabilities, corresponding to the 1000 classes in the ImageNet dataset. The input image is then assigned to the class with the highest probability.

Figure 5.1: VGG16 Architecture

First, we performed simple feature extraction by using the convolutional base of the network as is and training a new classifier on top of its output. We managed to obtain way better results w.r.t. our from scratch model.

Next, we decided to try to approach the fine tuning; initially we tried to unfreeze the last layer of the original convolutional base with the classifier trainable as well. Unfortunately we did not reliable results (see graphs below), because the network with an higher number of trainable parameters starts to overfit almost immediately.

So, we performed fine tune unfreezing once again the last layer but keeping the classifier frozen. Despite the reduction in terms of number of parameters we were not able to fix the previous issue. Because of that we decided to stop to keep unfreezing other layers as it would have resulted in worse performance.

5.1.1 Feature Extraction

Initially, we attempted to use the convolutional base of the architecture as a feature extractor only. Since we were working with paintings, which are vastly different from the images in ImageNet, we did not anticipate good results. Our expectations were misleading, as the model performed quite well since the first trials. In fact, experimenting with different combinations for the classifier and various regularization techniques, the model were able to perform reasonably well on both validation and test set.

The original VGG16 comes with a couple of 4096 FC layers followed by 1000 softmax neurons, which is alright for ImageNet but definitely oversized for our purpose. Hence, the convolutional base is left as it is, and the fully-connected block is replaced by a smaller version, followed by our output prediction layer made of 11 neurons. In this first experiment, we used a 2x dense layer with 512 neurons, followed by the output layer. The best result were obtained with LR = 0.0005.

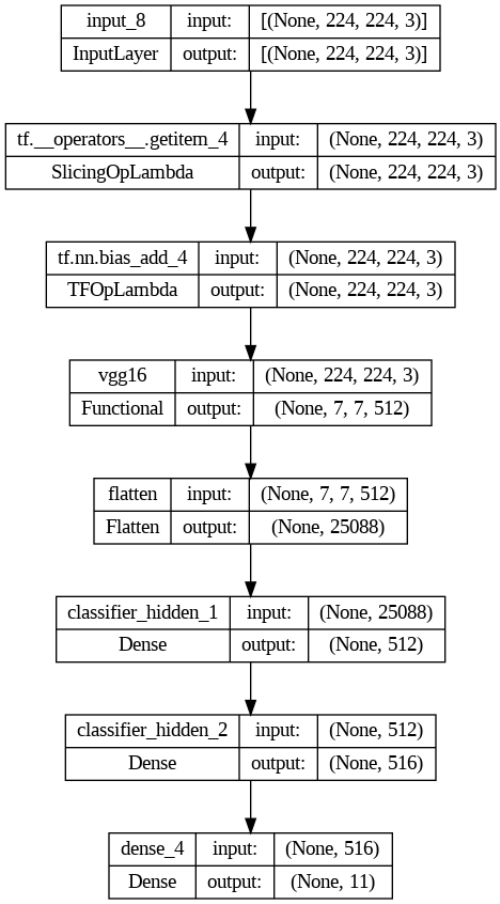


Figure 5.2: VGG16 with custom classifier structure

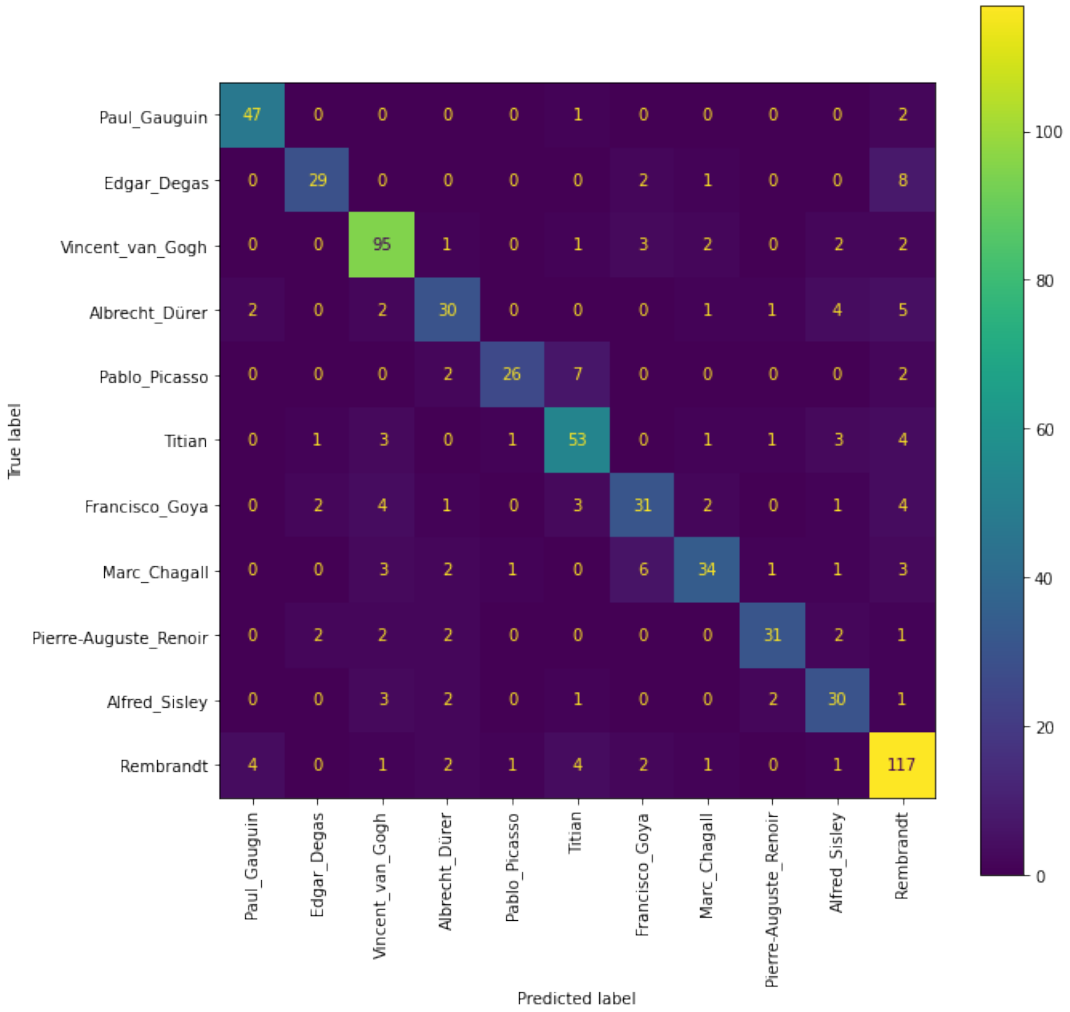
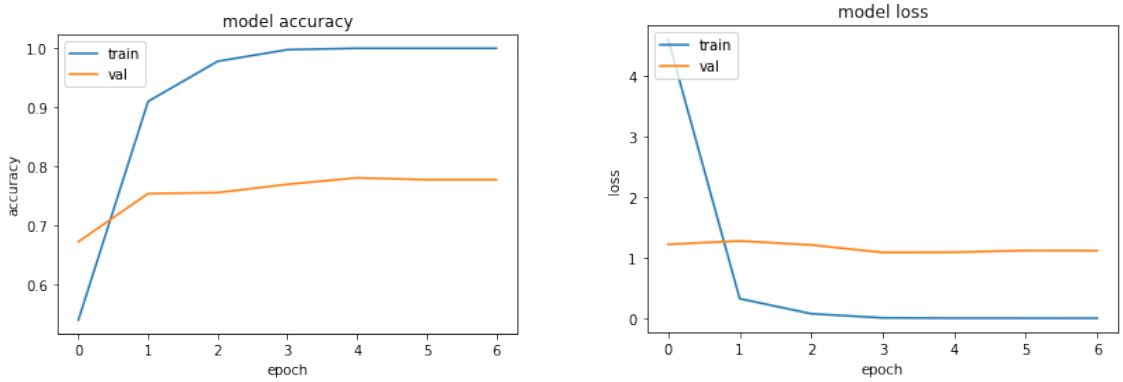


Figure 5.3: VGG16 with custom classifier, Confusion Matrix



(a) Model Accuracy

(b) Model Loss

Figure 5.4: Model History

Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Test Accuracy
0.006	0.99	1.08	0.77	0.8

This configuration seems to perform quite well in terms of accuracy, anyway, the results and the learning curves showed that the model was not able to generalize well. Specifically, while the training loss was 0.006, the validation loss was 1.08, which is quite high. The model clearly suffers from strong overfitting, as the training accuracy rapidly hits almost the 100% while the validation accuracy remains stuck at 80%. Even from the model loss plot we can observe a gap between train and validation losses.

5.1.2 Dropout

For addressing the issue of overfitting, we decided to add a Dropout layer. We set as dropout rate, which is the proportion of neurons in each layer that must be randomly chosen and temporarily removed; the removal of the neurons lasts only one epoch: in the following epoch, we perform again the removal using a different random set of neurons to be removed, while when we test the network we restore the original architecture of the network.

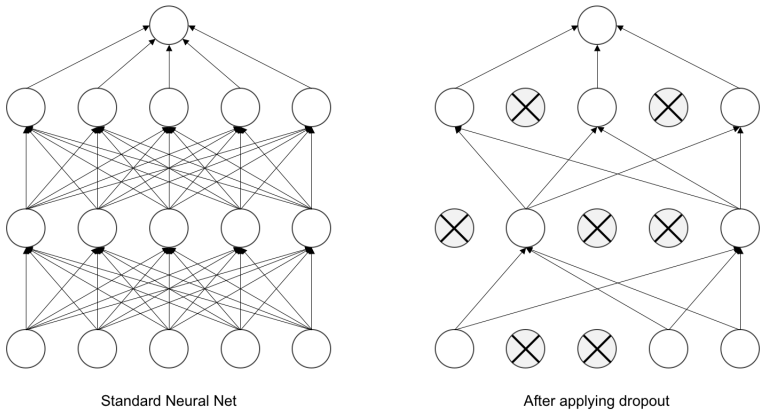


Figure 5.5: Dropout technique explained visually

The intuition of inserting the Dropout was linked to two theoretical aspects:

- **Minimizing Co-adaptation**: Dropout addresses a phenomenon that is called co-adaptation of neurons, for which each neuron cannot rely on the presence of the neighboring neurons to learn, since it doesn't know if the neighboring neuron will survive or not in the following epoch, so it is forced in some way to learn the really important features of the data that it receives as input, so the characteristics learned are more robust than the features that it could learn if it relies on the presence of the other neurons.
- **Increased Variability**: In a given epoch, we can transform the network as in the figure above, but in another epoch we can have a different set of active neurons, so in practice the network implements a concept referred as ensembling, that states that the result produced by the network are due to the average (or to an appropriate integration) of

the results produced by different networks, that are the subnetworks of the original networks produced by the dropout approach.

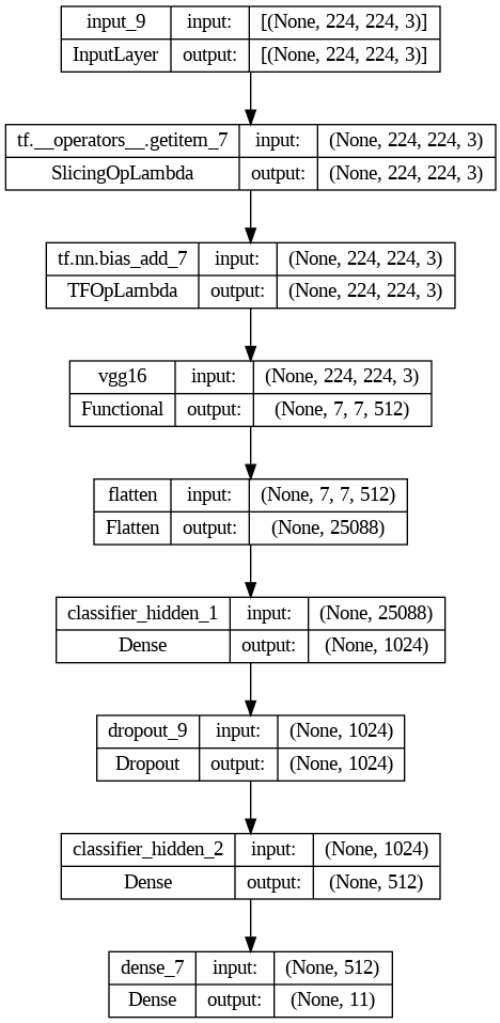


Figure 5.6: VGG16 with dropout layer structure

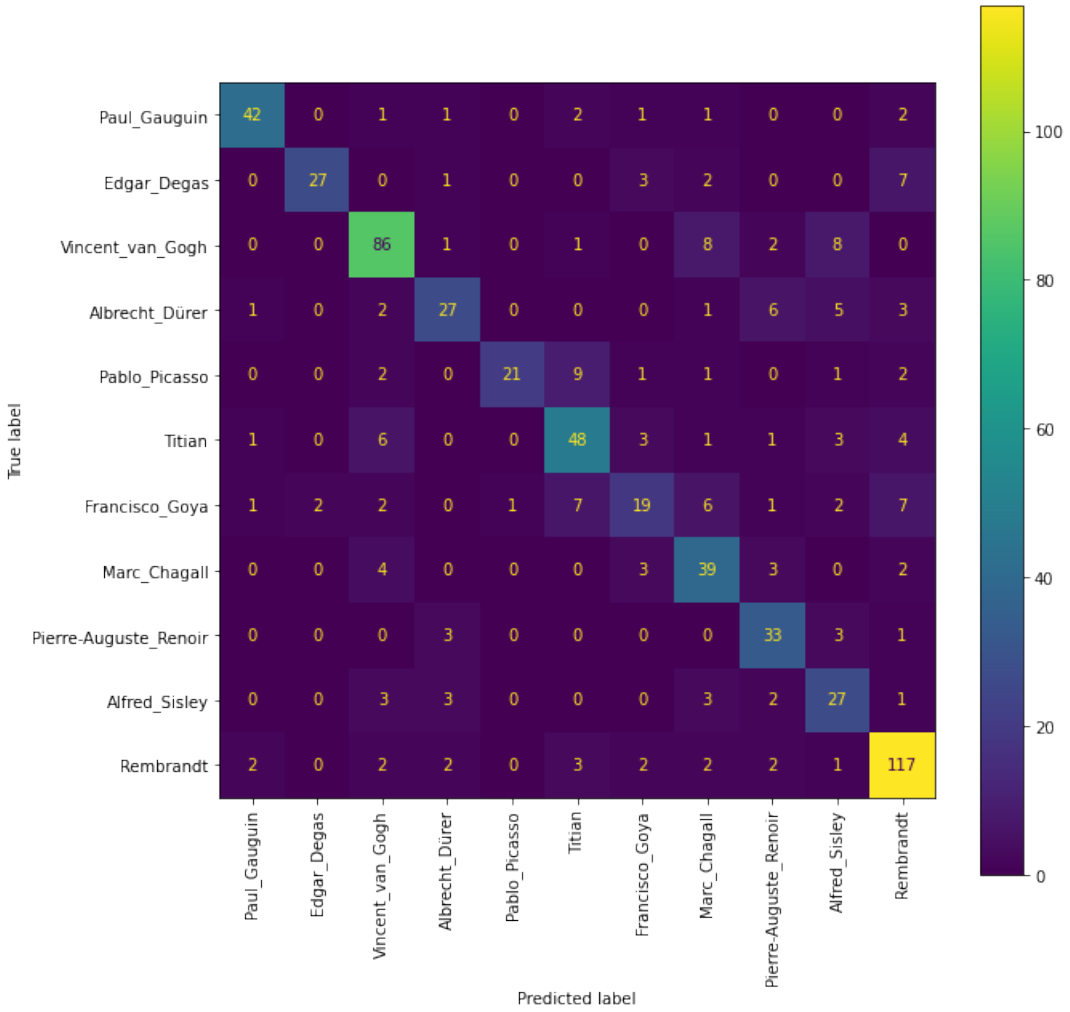
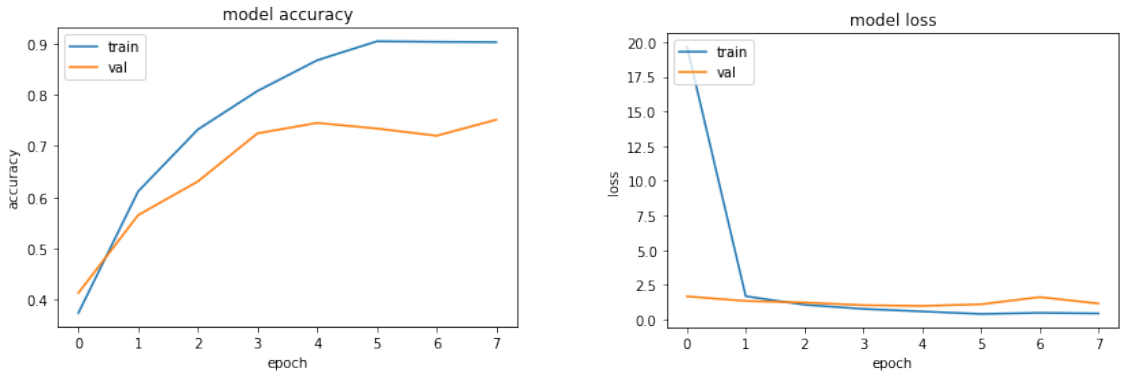


Figure 5.7: VGG16 with custom classifier, Confusion Matrix



(a) Model Accuracy

(b) Model Loss

Figure 5.8: Model History

As expected, dropout mitigated the magnitude of overfitting as we can see from the reduced

gap between the two accuracies. However our network perform slightly worst on accuracy than without the dropout layer.

Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Test Accuracy
0.57	0.86	0.97	0.74	0.74

Despite the better trend of the curves, the results in a decreased values of test accuracy (-0.03) and validation accuracy (-0.04). To note that we have still improvements thanks to the dropout, that reduces the validation loss considerably (-0.11).

5.1.3 Dropout, Batch Normalization

In addition to Dropout layer, we tried to add batch normalization layers, trying to combine both the effects of regularization, in order to tackle even more the overfitting and improve the generalization performance of the model.

While Dropout is a regularization technique that randomly drops out (i.e., sets to zero) some neurons during training, batch normalization is a technique that normalizes the activations of the previous layer, which can help stabilize the training process and improve the generalization performance of the network.
Using both dropout and batch normalization in a CNN can help achieve better results by combining the benefits of both regularization techniques. However, it is important to properly tune the hyperparameters (such as the dropout rate and the momentum parameter in batch normalization) to achieve optimal performance.

The intuition here came from looking at any of the hidden layers of our network: the activations from the previous layer are simply the inputs to the considered layer. Thus, the same logic that requires us to normalize the input for the first layer will also apply to each of these hidden layers. So, if we are able to somehow normalize the activations from each previous layer then the gradient descent will converge better during training. Batch Norm is just another network layer that gets inserted between a hidden layer and the next hidden layer. Its job is to take the outputs from the first hidden layer and normalize them before passing them on as the input of the next hidden layer.

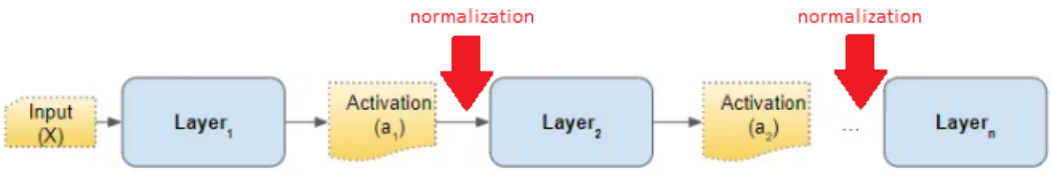


Figure 5.9: Batch Normalization explained visually

We found optimal the value of 0.4 as Dropout rate and 1e-4 for the batch normalization's momentum.

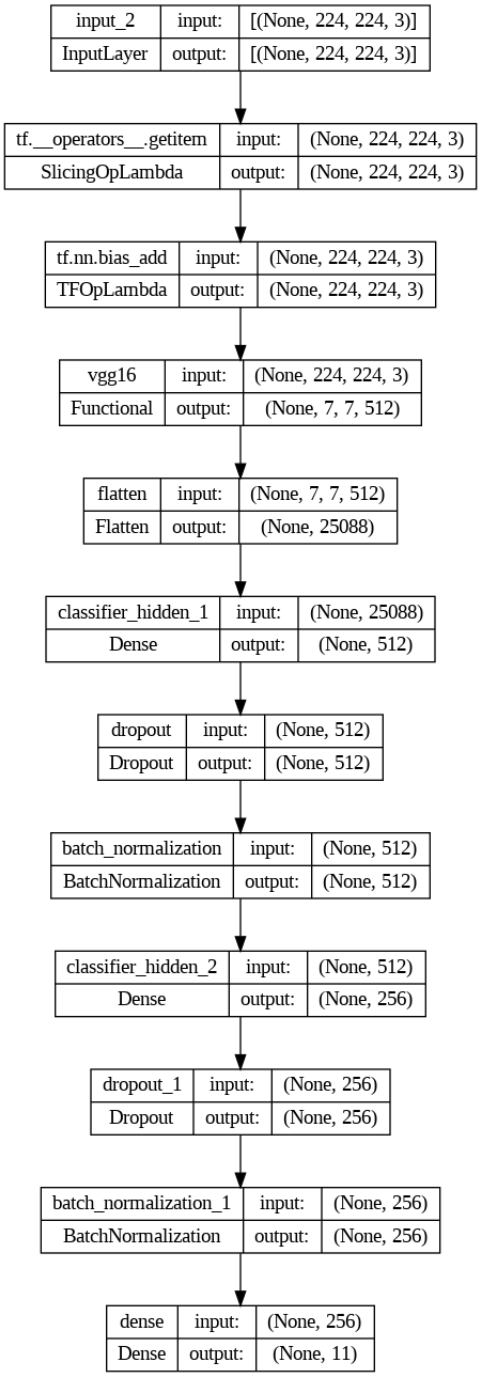


Figure 5.10: VGG16 with both batch normalization and dropout layers, structure

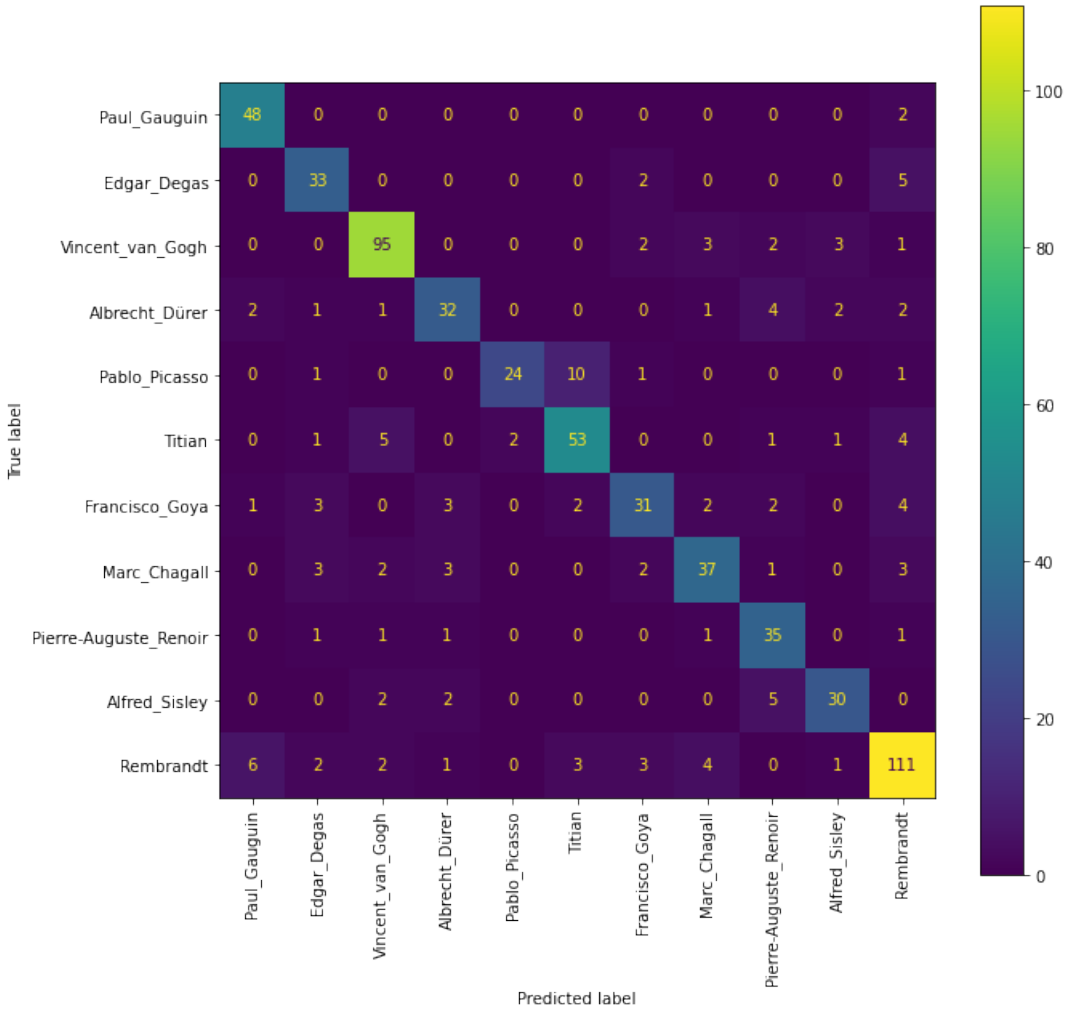
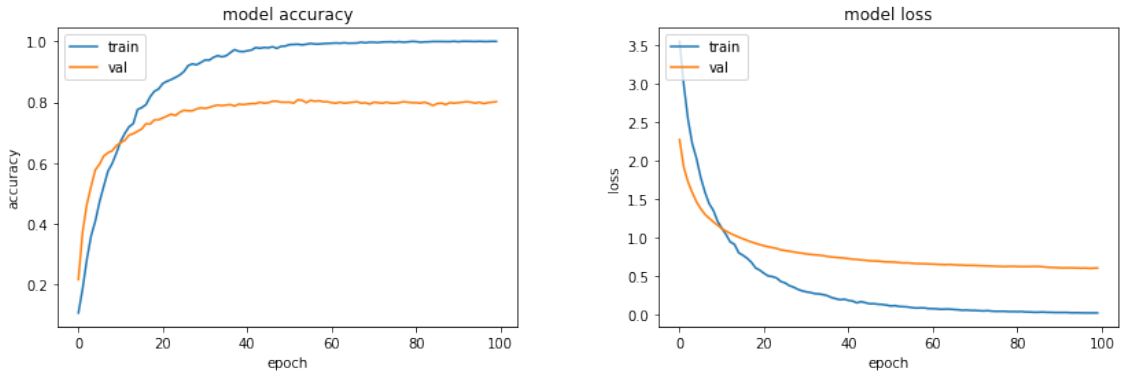


Figure 5.11: VGG16 with dropout + batch normalization, Confusion Matrix



(a) Model Accuracy

(b) Model Loss

Figure 5.12: Model History

The Batch Normalization helped massively in the trend of the learning curve. In fact we

were able to reach a great reduction of the gap between train and validation accuracies, with a more stable and less oscillating trend. Moreover, the losses dramatically decreases, both going under 0.65.

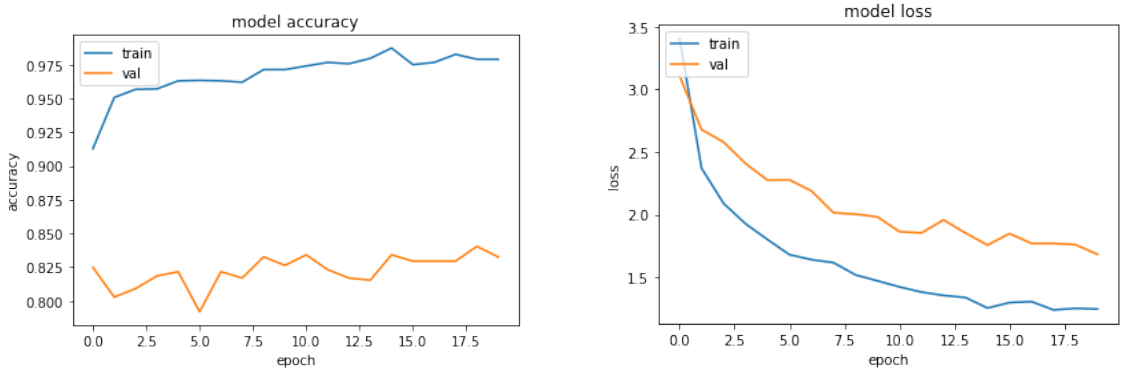
Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Test Accuracy
0.02	0.99	0.60	0.80	0.80

The results have improved a lot with the help of Batch Normalization in all the metrics. Specifically in test set accuracy (+0.06), validation accuracy (+0.06), validation loss (-0.37). This represents the best results we could achieve with VGG16 model so far.

5.1.4 Fine Tuning

Using the model defined in section 5.1.1 we started by un-freezing the 3rd Conv2D layer in the 5th block (last Conv2D layer) and than retrained the network.

The following result are obtained using Adam as optimizer with LR set to 0.0001, 20 epochs and patience for early stopping equal to 5:



(a) Model Accuracy

(b) Model Loss

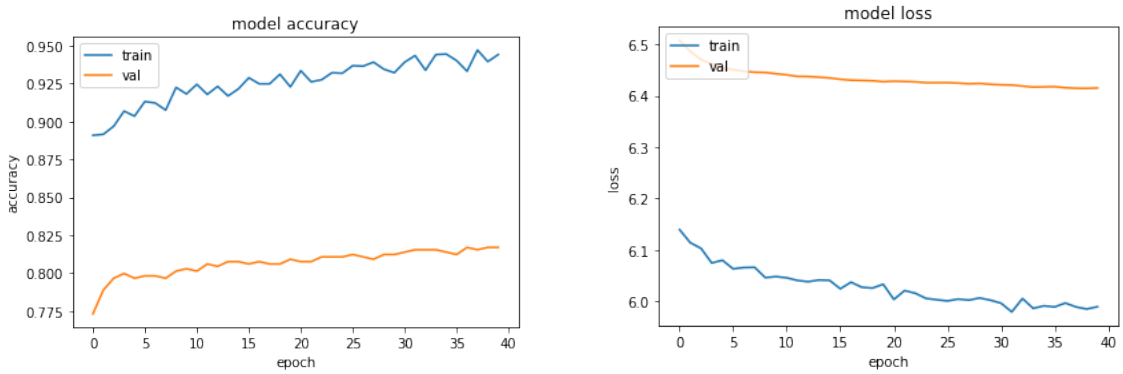
Figure 5.13: Model History

As we can observe by the plots, unfreezing an entire convolutional layer of the original VGG network results in a serious overfitting issue. In fact the training and validation accuracies do not follow an increasing trend (as it should be) but they start immediately from a very high value (91% on train, 82% on validation).

This phenomenon could be caused by the enormous amount of trainable parameters (28.581.387) compared to the size of our available training set (just 3000 samples). The effects of this gap end up with the network to just learn almost perfectly the features of the samples and lose the ability to generalize.

This issue forced us to stop the fine tuning approach, as if we would unfreeze other layers it would only get worse because the number of trainable parameters would increase as well. So, we tried to train (unfreeze) only the last layer of the convolutional base and to freeze our

custom classifier. Moreover, we decreased the learning rate to 0.000001, in order to let the curves learn more slowly.



(a) Model Accuracy

(b) Model Loss

Figure 5.14: Model History

Neither freezing the classifier nor the decreased LR helped fixing the above mentioned issue, so we end up once again that the problem is related to the dataset size.

5.2 ResNet50

ResNet-50 is a 50-layer deep convolutional neural network (CNN) that was created to address the issue of accuracy degradation in deep neural networks. It uses residual blocks, which employ "skip connections" to allow for alternate paths for the gradient to flow through, resulting in improved accuracy. The architecture of ResNet-50 includes five convolutional blocks with non-decreasing filter sizes, followed by a dense classifier and a final 1000-node softmax layer. The network takes in an image of fixed size (224x224x3) and outputs a vector of 1000 probabilities for the 1000 classes in the ImageNet dataset, with the image being assigned to the class with the highest probability. ResNet also has other variants with different number of layers.

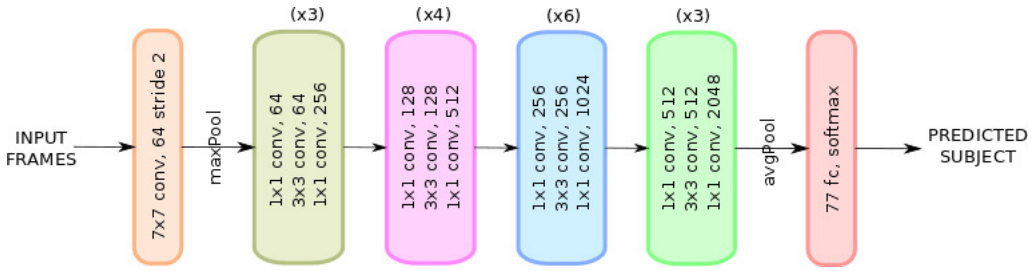


Figure 5.15: ResNet50 Architecture

Again, we conducted our experiments in two stages. First, we began with basic feature extraction by using the convolutional base of the network and adding new classifiers on top of its output. Next, we selected the best models and attempted to fine-tune the convolutional

base by incrementally unfreezing its layers. However, the results were not satisfactory, mainly due to the small and dissimilar nature of our dataset compared to ImageNet. The conclusions were similar to VGG experiments for fine tuning.

5.2.1 Feature Extraction

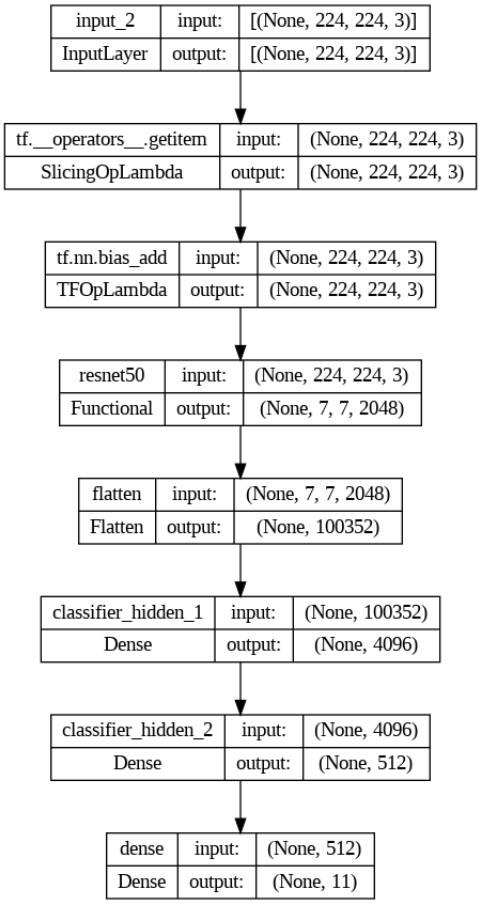


Figure 5.16: ResNet with custom classifier structure

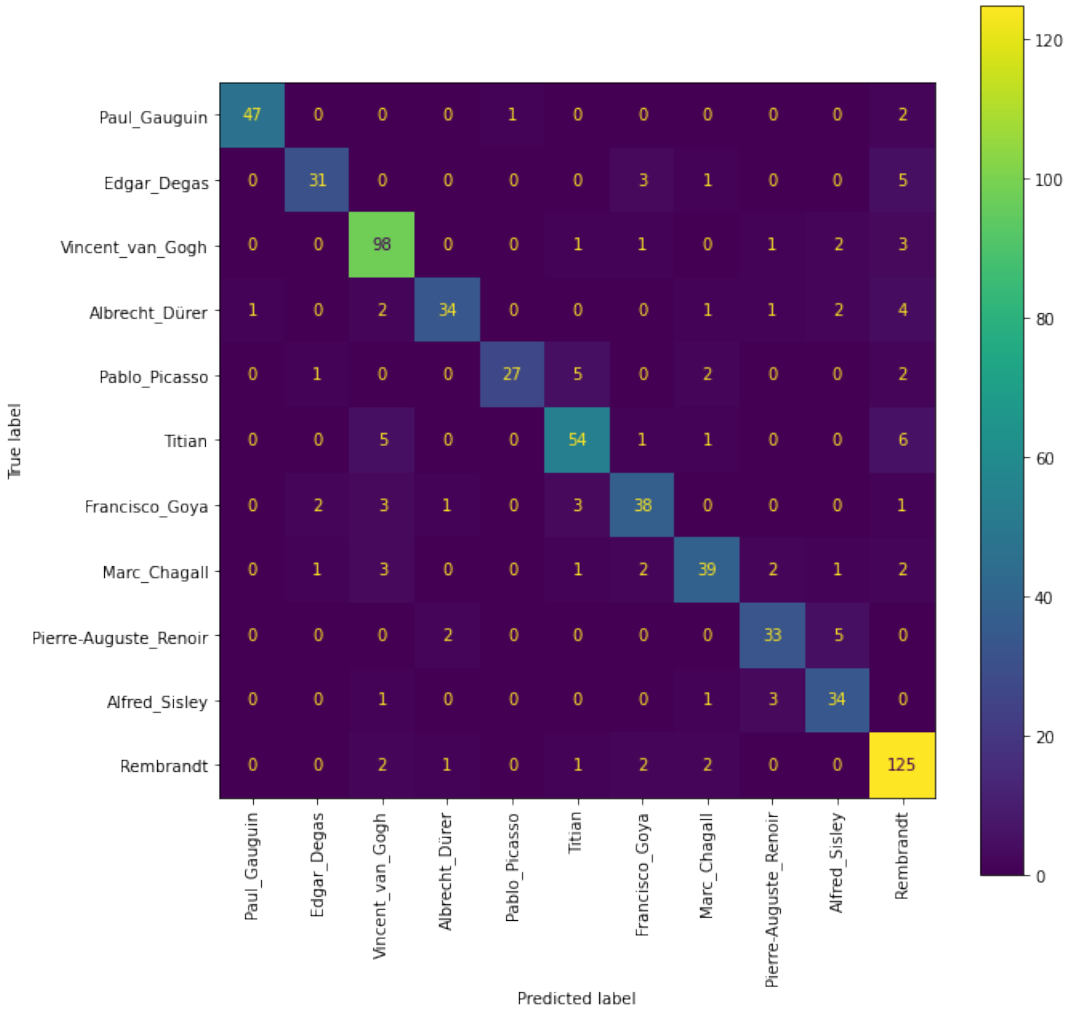
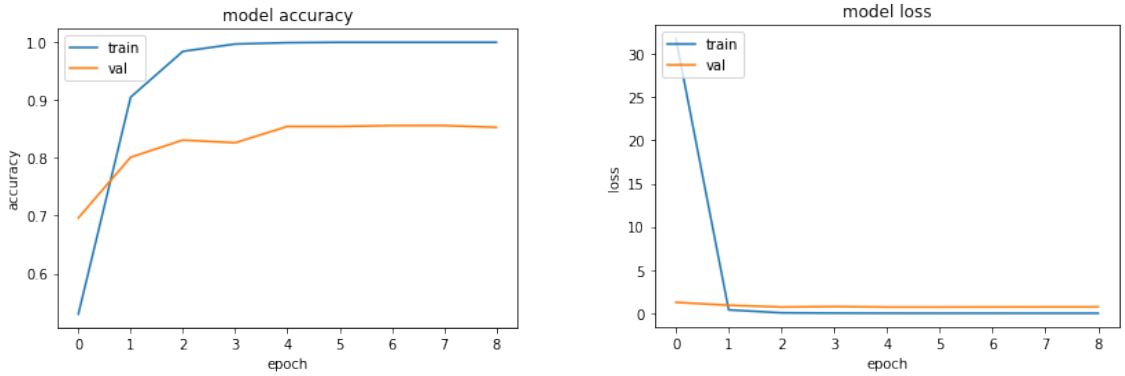


Figure 5.17: ResNet with custom classifier, Confusion Matrix

As we immediately observe the missclassified samples are significantly less w.r.t. to our from scratch model. This is may due to the fact that the convolutional base of resnet can rely on a much larger image dataset.



(a) Model Accuracy

(b) Model Loss

Figure 5.18: Model History

As we can see from the Accuracy graph the train rapidly goes towards 1, whereas the validation accuracy remains stuck at 80%, this is a clear sign of overfitting.

Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Test Accuracy
0.0016	1.0	0.79	0.83	0.82

The network is able to perform well both on test set and validation set reaching the best result so far. Also to be mentioned, the very low loss values.

5.2.2 Dropout

In order to try to mitigate the overfitting problem, we added two dropout layers as described in the following image.

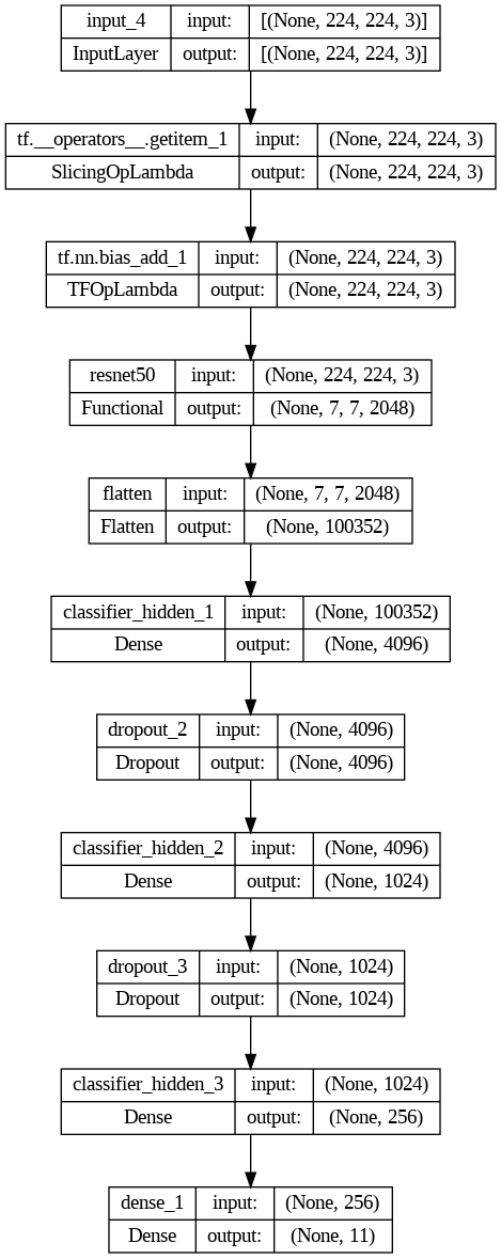


Figure 5.19: ResNet with dropout, Structure

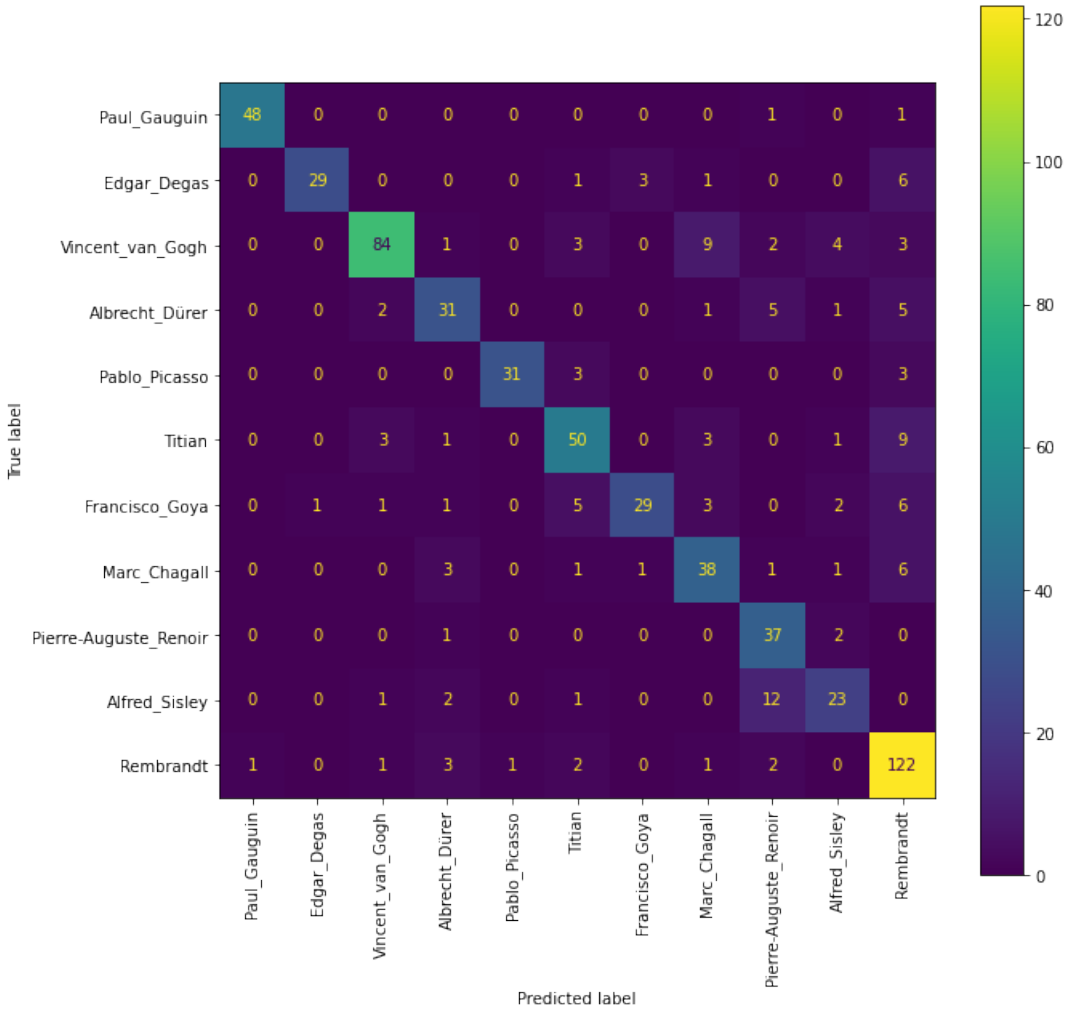
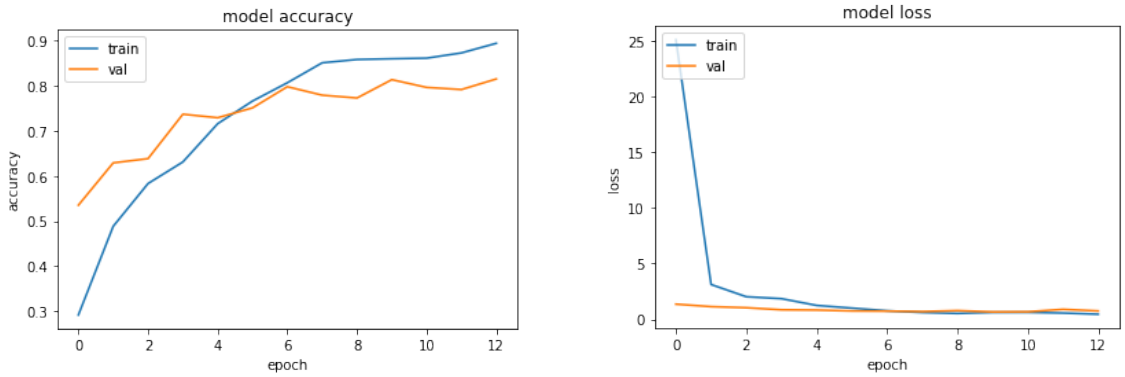


Figure 5.20: ResNet with dropout, Confusion Matrix



(a) Model Accuracy

(b) Model Loss

Figure 5.21: Model History

As expected the accuracy trends are similar and less steep, moreover the gap between them is significantly reduced. Anyway, we experienced a reduction in terms of validation and training accuracies.

Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Test Accuracy
0.62	0.86	0.67	0.81	0.79

The model underperforms w.r.t. the previous experiment; specifically in test accuracy (-0.03), validation accuracy (-0.02) and both the errors. Anyway, once again the dropout layers help reducing the loss of the validation set (-0.12).

5.2.3 Dropout, Batch Normalization

As for VGG16, we tried to add batch normalization layers, trying to combine both the effects of regularization, in order to tackle even more the overfitting and improve the generalization performance of the model.

Again, we found optimal the value of 0.4 as Dropout rate and 1e-4 for the batch normalization's momentum.

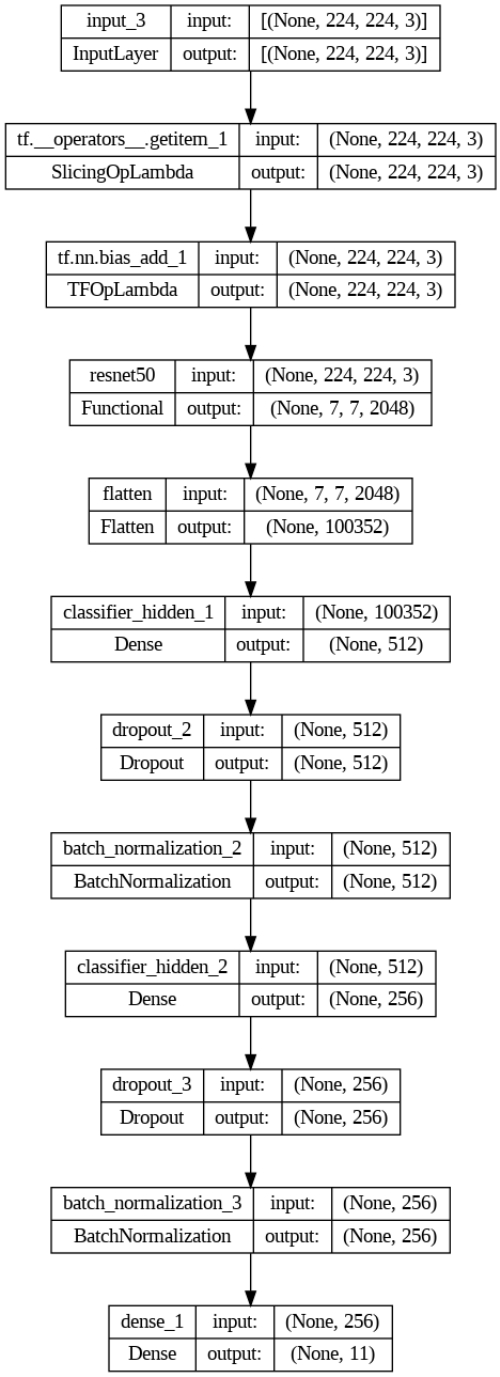


Figure 5.22: ResNet with both batch normalization and dropout layers, structure

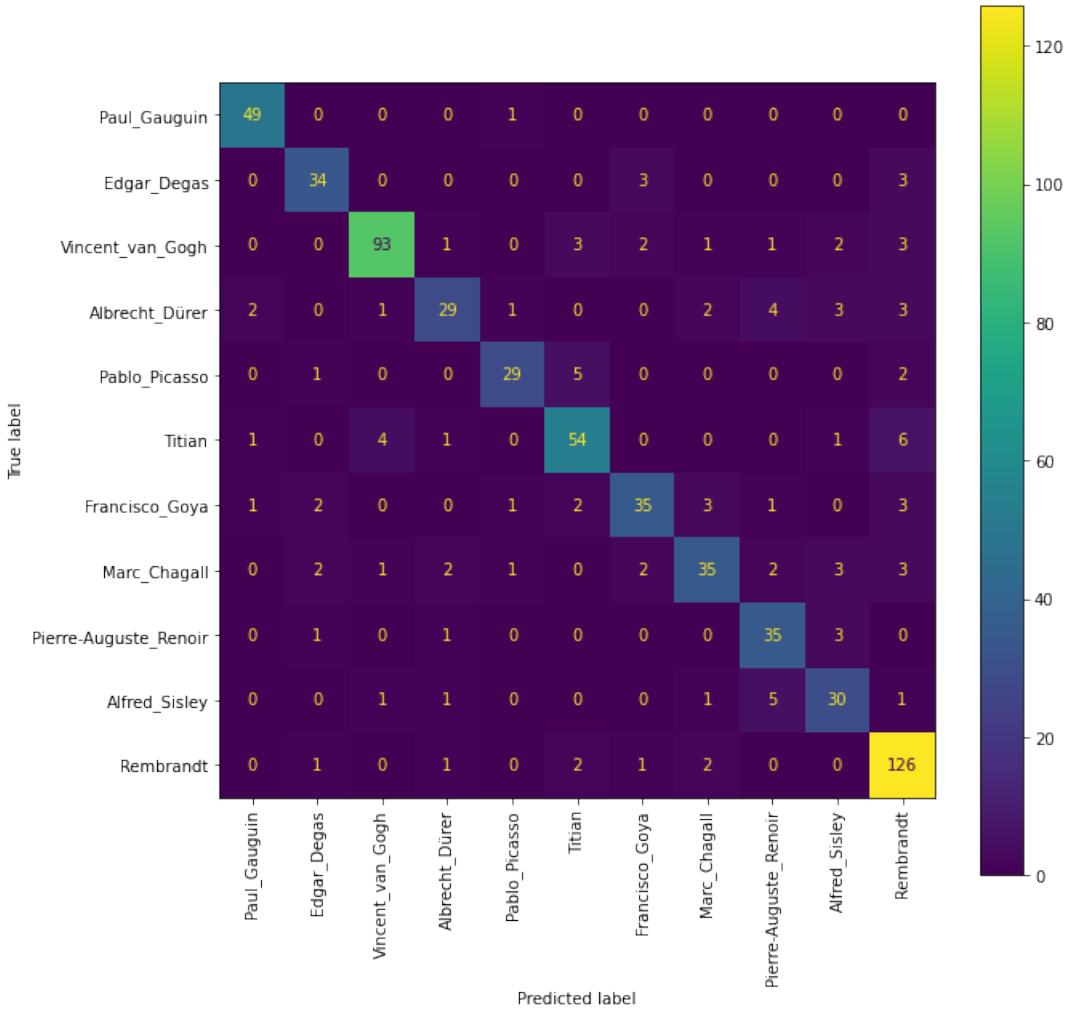
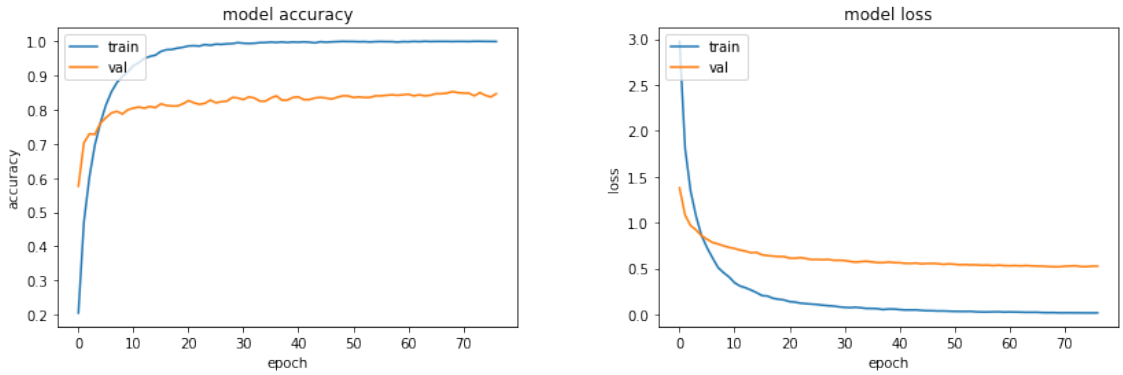


Figure 5.23: ResNet with dropout + batch normalization, Confusion Matrix



(a) Model Accuracy

(b) Model Loss

Figure 5.24: Model History

Also in this model the Batch Normalization helped in the trend of the learning curve. In

fact we were able to reach a great reduction of the gap between learning accuracies, with a more stable trend. Moreover, the losses greatly decreases, both going under 0.55.

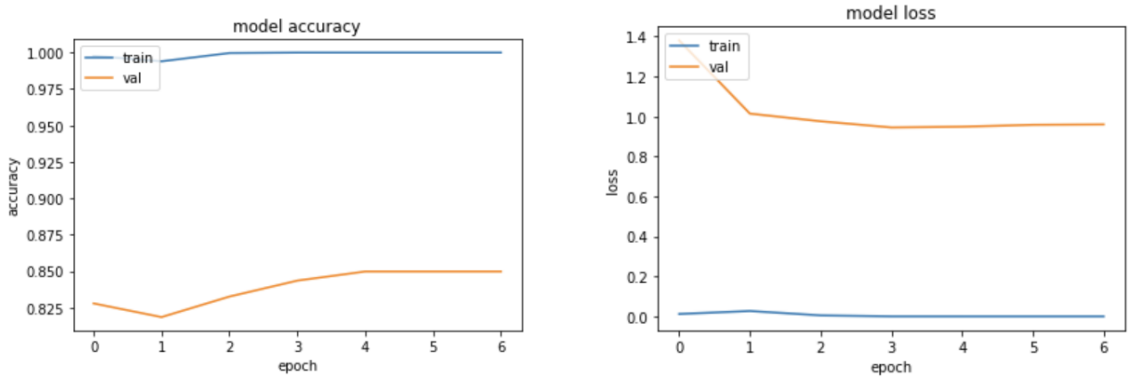
Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Test Accuracy
0.01	0.99	0.51	0.85	0.83

The results have improved a lot with the help of Batch Normalization in all the metrics. Specifically in test set accuracy (+0.04), validation accuracy (+0.04), validation loss (-0.16). This represents the best results we could achieve with ResNet model so far.

5.2.4 Fine Tuning

Using the model defined in section 5.2.1 we started by un-freezing the last convolutional layer of the last block and then retrained the network.

The following result are obtained using Adam as optimizer with LR set to 0.0001, 20 epochs and patience for early stopping equal to 5:



(a) Model Accuracy

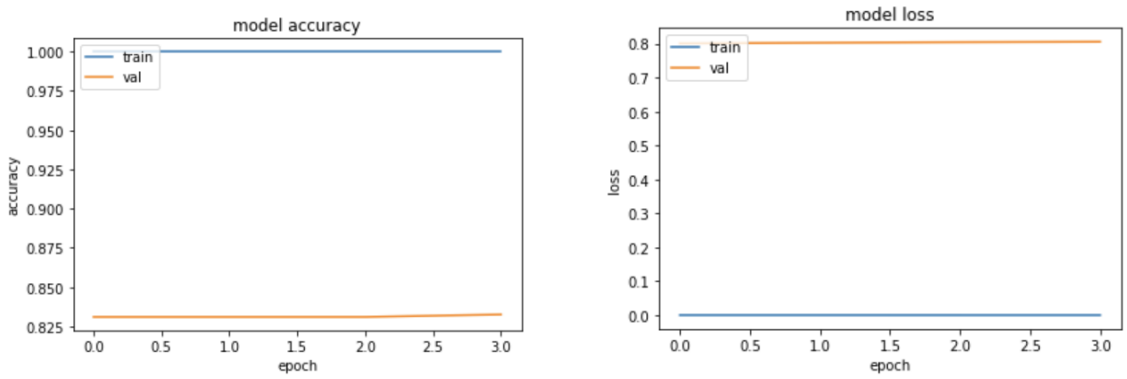
(b) Model Loss

Figure 5.25: Model History

As we can observe by the plots, unfreezing an entire convolutional layer of the original ResNet50 network results in a serious overfitting issue. In fact the training and validation accuracies do not follow an increasing trend (as it should be) but they start immediately from a very high value (99% on train, 82% on validation).

This phenomenon could be caused by the enormous amount of trainable parameters, more than 400 million, compared to the size of our available training set (just 3000 samples). The effects of this gap end up with the network to just learn almost perfectly the features of the samples and lose the ability to generalize.

Then we tried to train (unfreeze) only the last layer of the convolutional base and to freeze our custom classifier. Moreover, we decreased the learning rate to 0.000001, in order to let the curves learn more slowly.



(a) Model Accuracy (b) Model Loss

Figure 5.26: Model History

Neither freezing the classifier nor the decreased LR helped fixing the above mentioned issue, so we end up once again that the problem is related to the dataset size.

5.3 InceptionV3

InceptionV3 is a deep convolutional neural network (CNN) architecture developed for image classification. It is part of the Inception network family and was trained on a large dataset (ImageNet) to identify objects within an image. InceptionV3 is known for its efficient use of computation and parameters, making it a popular choice for image classification tasks. The network architecture is characterized by its Inception modules, which are made up of multiple parallel convolutional and pooling layers that extract features at multiple scales.
The InceptionV3 architecture is designed to extract features from the input image at multiple scales and to combine them in a way that effectively represents the objects in the image. The entire network has approximately 23 million parameters, making it a relatively lightweight model that can be efficiently trained on large datasets.

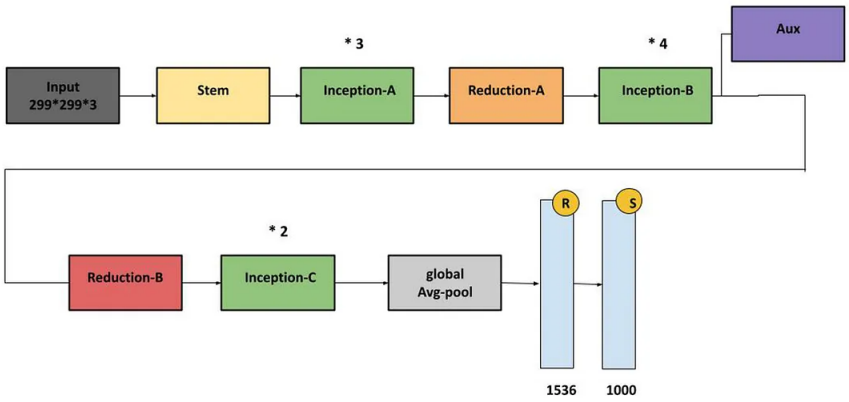


Figure 5.27: Inception with custom classifier structure

Also for this pre trained model, we started with basic feature extraction by utilizing the convolutional base of the network and incorporating new classifiers on top of the output. Subsequently, we chose the most suitable models and tried to refine the convolutional base by gradually unfreezing its layers. Unfortunately, the results were unsatisfactory, mainly due to the limited and dissimilar characteristics of our dataset compared to ImageNet.

5.3.1 Feature Extraction

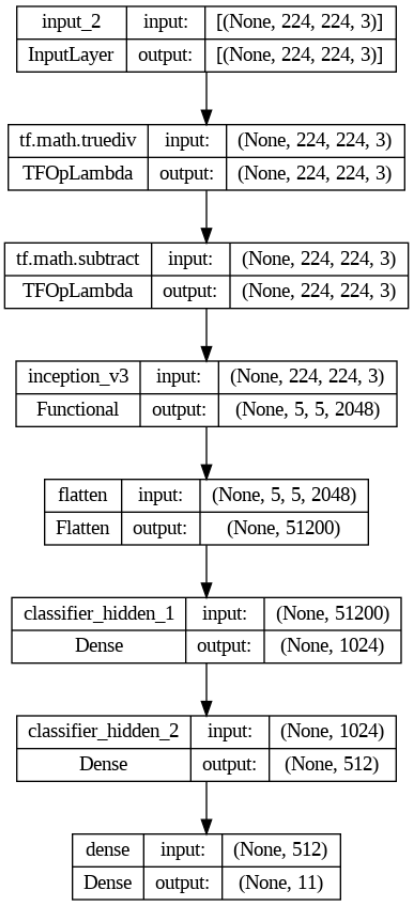


Figure 5.28: Inception with custom classifier structure

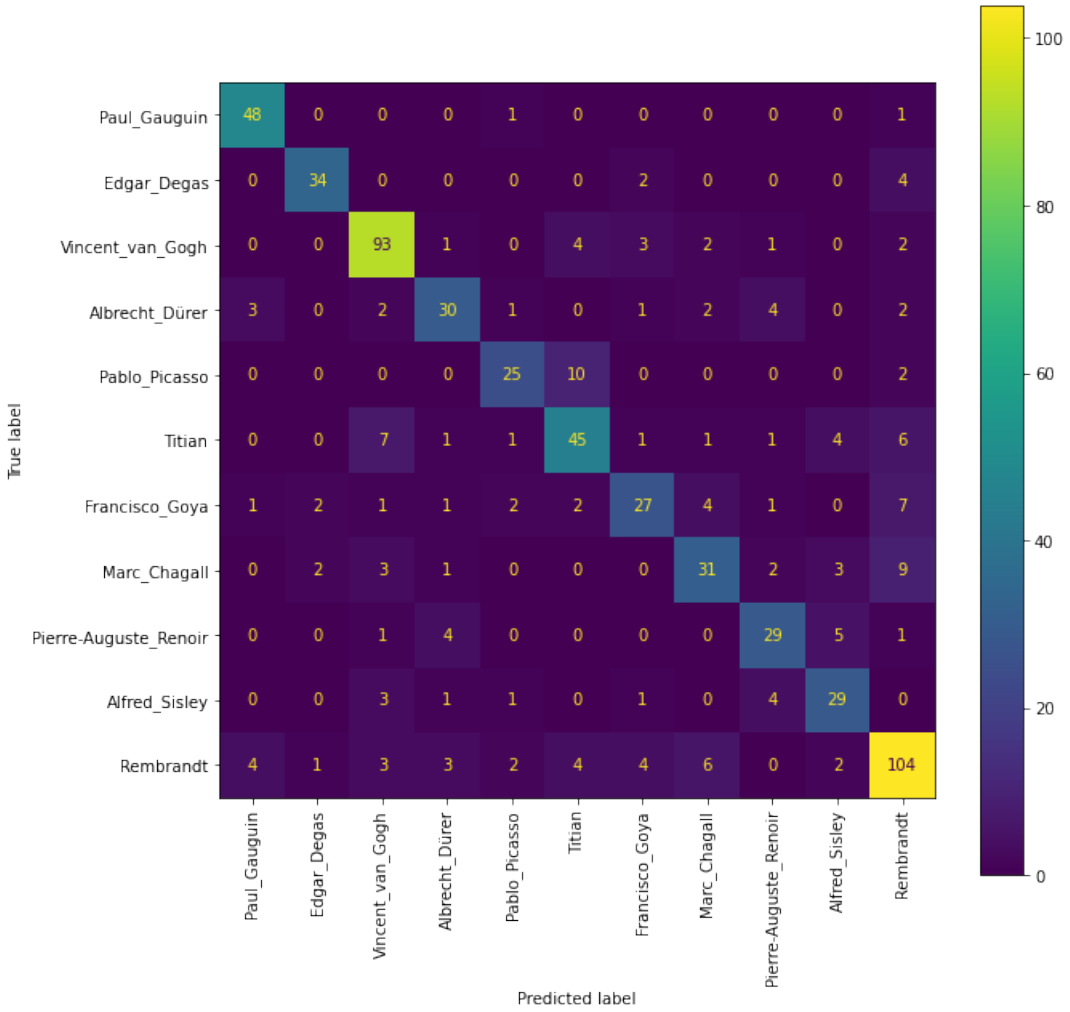
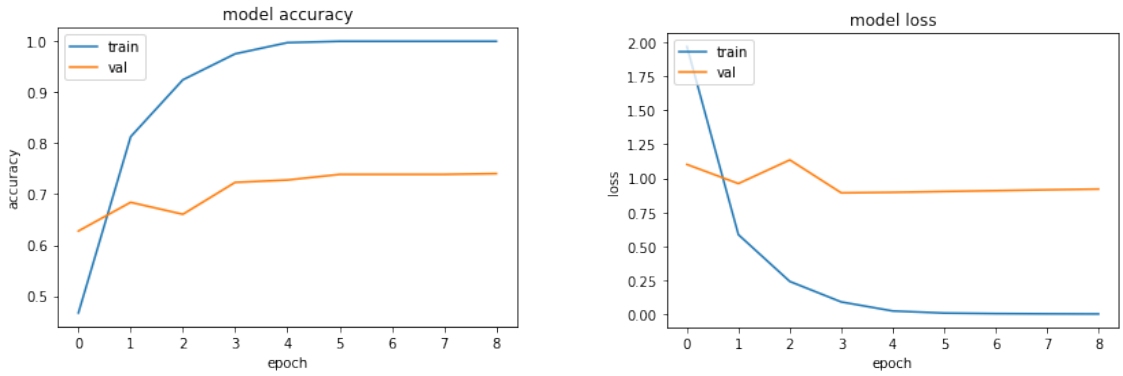


Figure 5.29: Inception with custom classifier, Confusion Matrix



(a) Model Accuracy

(b) Model Loss

Figure 5.30: Model History

Similarly to ResNet with feature extraction, we experienced a steep training accuracy curve, fastly reaching the value of 1. There also is the overfitting problem since the gap is quite large.

Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Test Accuracy
0.09	0.97	0.89	0.72	0.75

The results were not as good as in the previous models, but overall they are acceptable.

5.3.2 Dropout

Again to prevent the model to overfit too quickly we exploited one dropout layer.

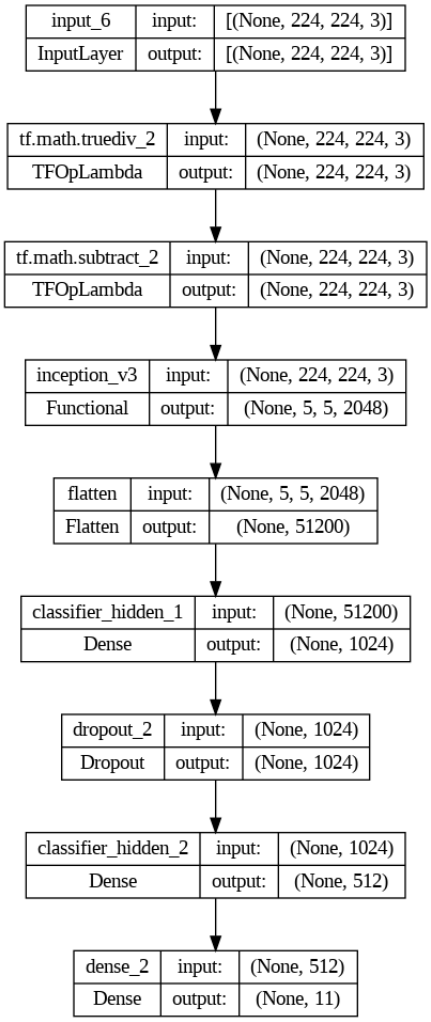


Figure 5.31: Inception with custom classifier structure

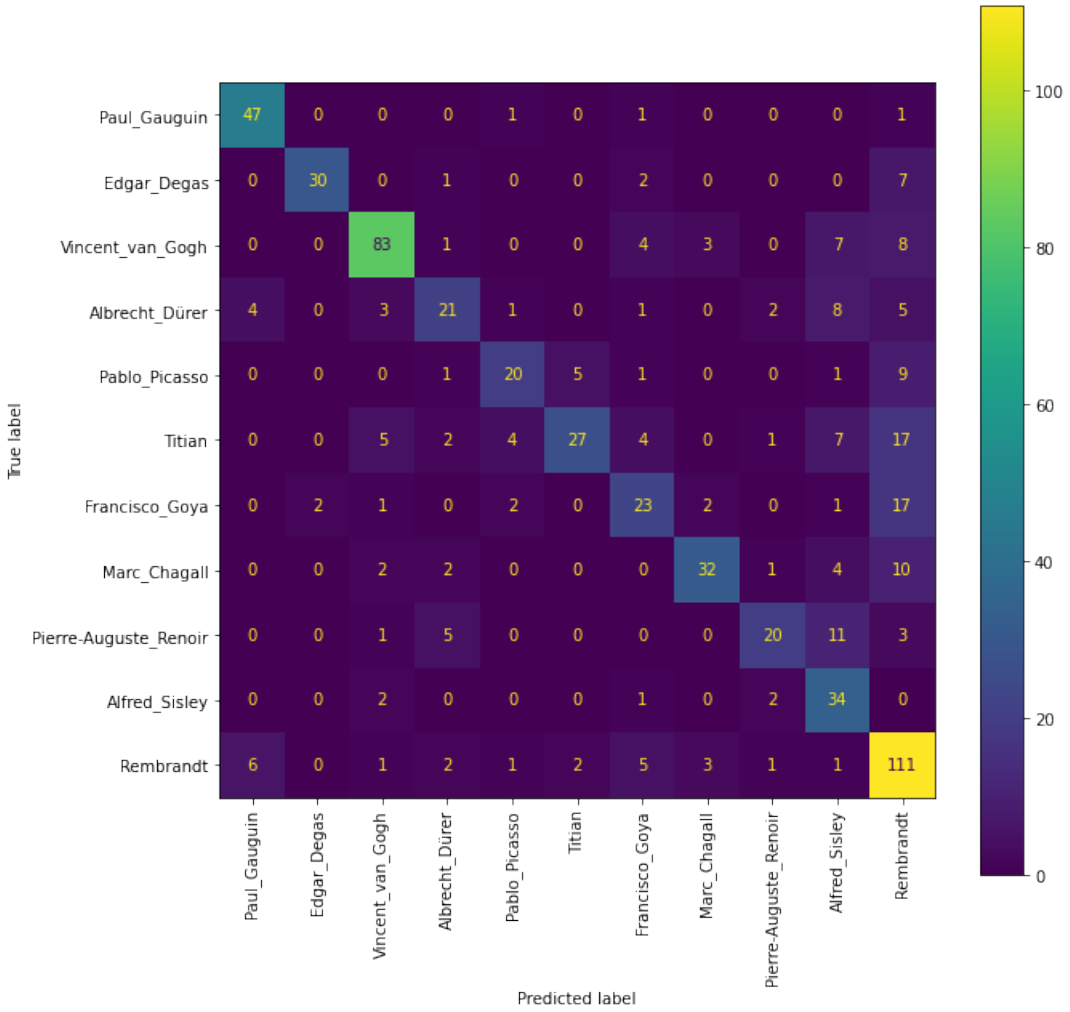
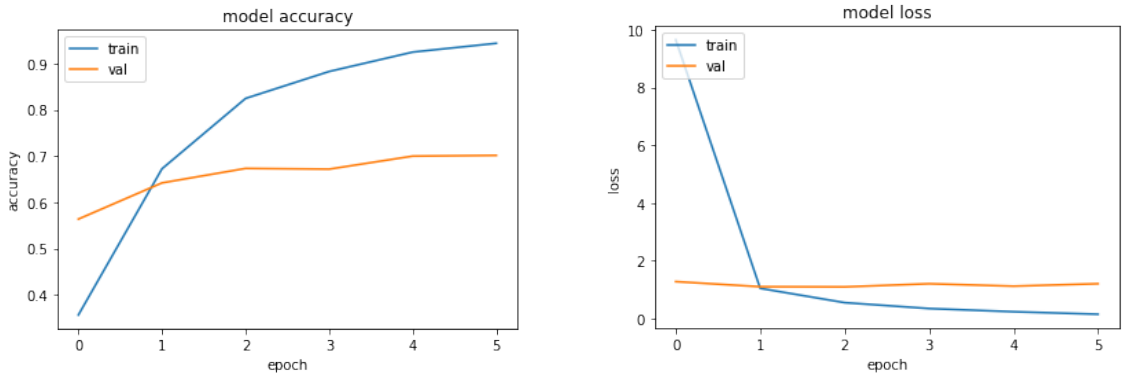


Figure 5.32: Inception with custom classifier, Confusion Matrix



(a) Model Accuracy

(b) Model Loss

Figure 5.33: Model History

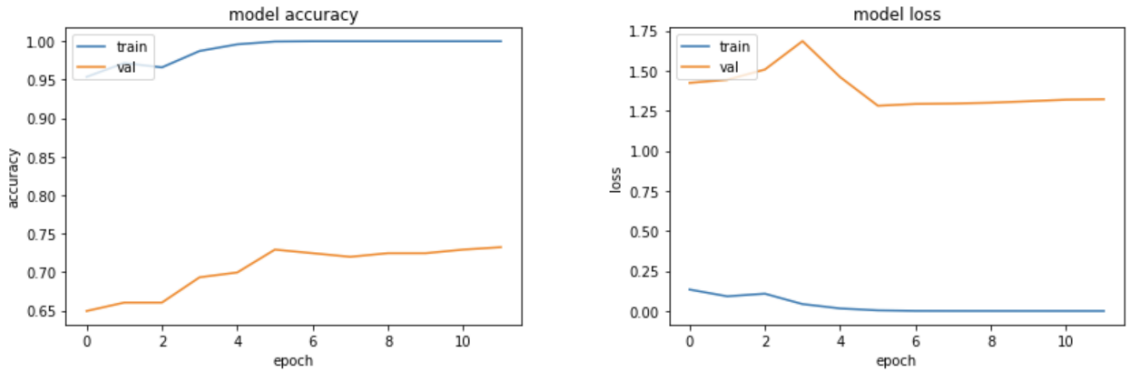
Training Loss	Training Accuracy	Validation Loss	Validation Accuracy	Test Accuracy
0.54	0.82	1.09	0.67	0.68

Once again the dropout layer lowered the results.

5.3.3 Fine Tuning

Using the model defined in section 5.3.1 we started by un-freezing the last convolutional layer ("conv2d_93") and than retrained the network.

The following result are obtained using Adam as optimizer with LR set to 0.0001, 20 epochs and patience for early stopping equal to 6:



(a) Model Accuracy

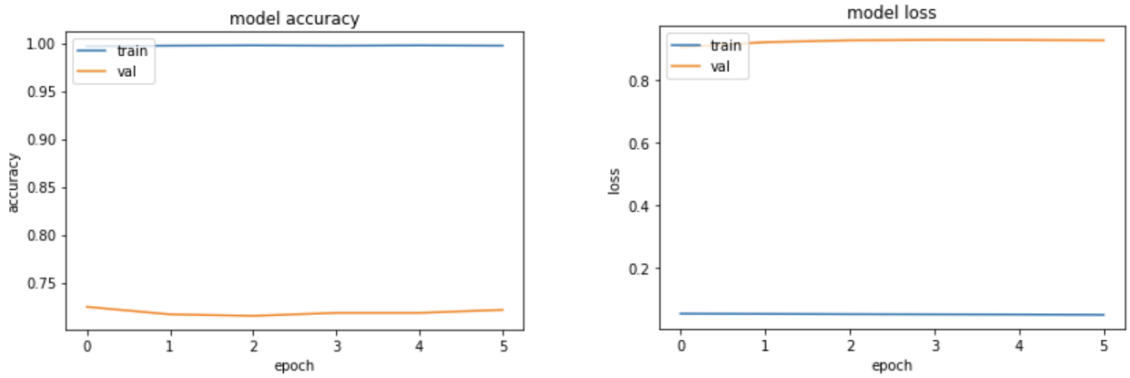
(b) Model Loss

Figure 5.34: Model History

As we can observe by the plots, unfreezing an entire convolutional layer of the original Incepeption_v3 network results in a serious overfitting issue.

This phenomenon could be caused by the enormous amount of trainable parameters (about 50 millions) compared to the size of our available training set (just 3000 samples). The effects of this gap end up with the network to just learn almost perfectly the features of the samples and lose the ability to generalize.

Then we tried to train (unfreeze) only the last layer of the convolutional base and to freeze our custom classifier. Moreover, we decreased the learning rate to 0.000001, in order to let the curves learn more slowly.



(a) Model Accuracy (b) Model Loss

Figure 5.35: Model History

Neither freezing the classifier nor the decreased LR helped fixing the above mentioned issue, so we end up once again that the problem is related to the dataset size.

Chapter 6

Ensemble

Ensembling is a technique where multiple models' predictions are combined to create improved results. This approach is based on the idea that diverse, high-performing models that are trained separately will have different perspectives on the data, capturing different aspects of the truth. By bringing these perspectives together, a more accurate understanding of the data can be achieved.
The use of ensembles is prevalent in modern machine learning contests, especially on Kaggle, where the winners typically employ large ensembles of models. This approach outperforms any single model, even if it is highly effective, leading us to decide to incorporate it in our project.

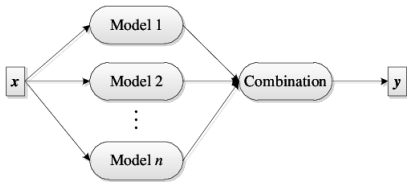


Figure 6.1: General Ensemble Architecture

We merged together four different models:

- VGG16
- ResNet
- InceptionNet
- our best network from section 3.8.2

We decided to concatenate the predicted labels and to compute the mean among them. Then we can generate the confusion matrix for the ensembled predictions.

```
Accuracy on test set: 0.8597560975609756
                      precision    recall  f1-score   support

         PaulGauguin     0.9434    1.0000    0.9709        50
          EdgarDegas     0.9000    0.9000    0.9000        40
     VincentvanGogh     0.8559    0.9528    0.9018       106
       AlbrechtDurer     0.9000    0.6000    0.7200        45
        PabloPicasso     1.0000    0.8378    0.9118        37
              Titian     0.8966    0.7761    0.8320        67
       FranciscoGoya     0.7872    0.7708    0.7789        48
         MarcChagall     0.8182    0.7059    0.7579        51
Pierre-AugusteRenoir     0.7609    0.8750    0.8140        40
        AlfredSisley     0.7727    0.8718    0.8193        39
           Rembrandt     0.8621    0.9398    0.8993       133

            accuracy                         0.8598       656
           macro avg     0.8634    0.8391    0.8460       656
        weighted avg     0.8631    0.8598    0.8570       656
```

Listing 6.1: Classification result over the test set

The enseble network outperforms every previous model, being the best experiment so far. Specifically it is able to increase the accuracy on test set of about 5% w.r.t. the model obtained in section 5.2.1.

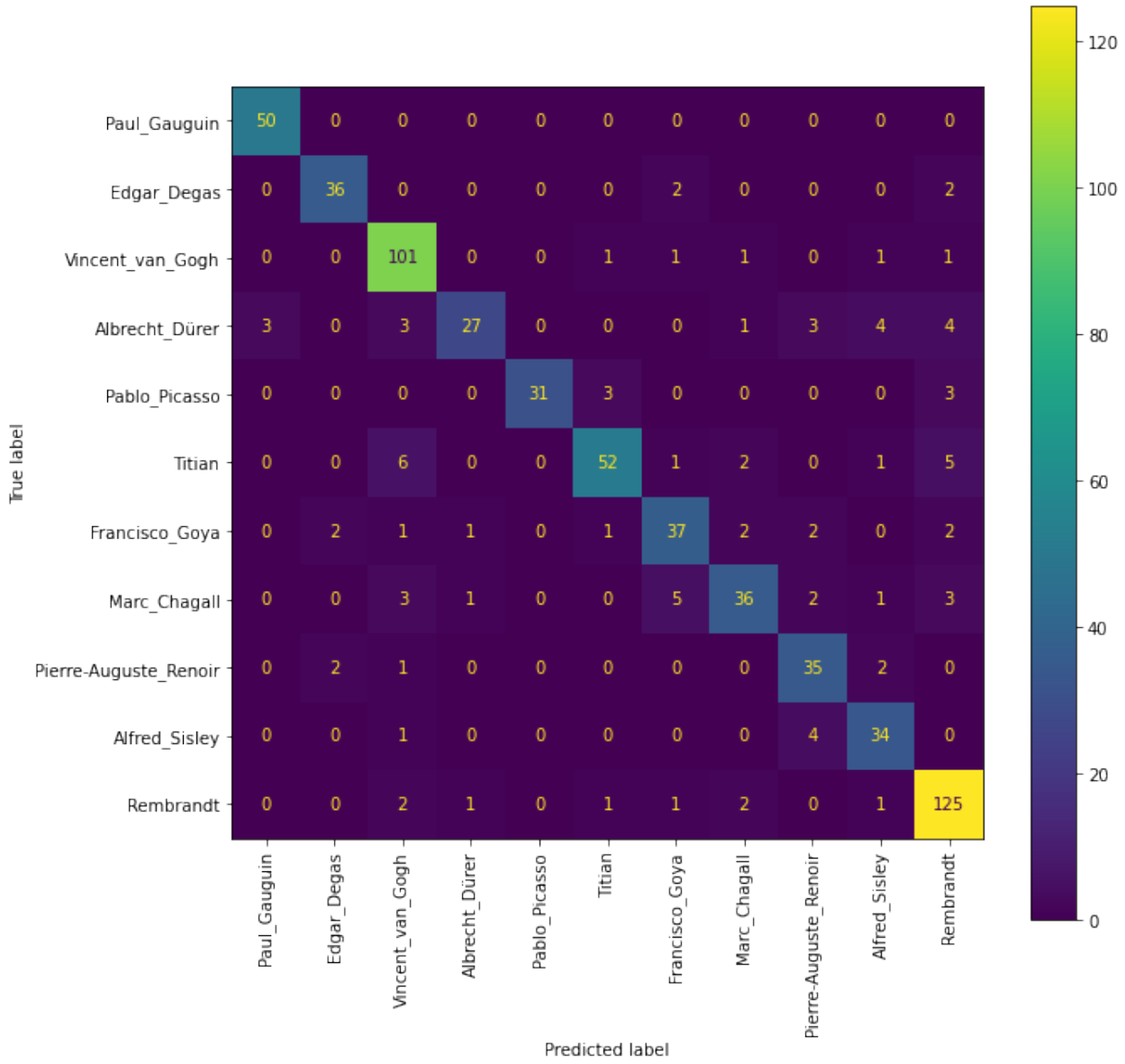


Figure 6.2: Ensemble Confusion Matrix

We can observe that in the resulting confusion matrix the number of missclassified labels is significantly lower w.r.t. our from scratch model. This reflects in the test accuracy, that is the highest we managed to reach.

6.1 Genetic Algorithm

We took a step further, investigating the possibility of applying a custom defined genetic algorithm to select the best combination of weights for computing the weighted average of the predictions for the ensemble model. This intuition comes from the necessity to assign a different weight to each model because they do not perform in the same manner. Instead of

choosing statically the set of weights, we exploit the power of a genetic algorithm to choose the optimal combination.

Genes

A single gene corresponds to a single weight assigned to a model

Chromosome:

is a set of four genes, thus four weights assigned accordingly to the models in the previous bullet list. Each Chromosome is normalized so that the sum of the weights (genes) equals to 1.

Fitness Function:

corresponds to the accuracy of the ensemble model

Mutation:

we implemented the Swap Mutation with probability 0.3. The choice of a low probability is due to the fact that an high one brings to a random search.

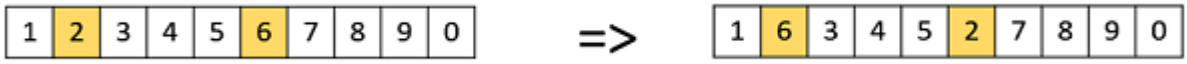


Figure 6.3: Swap Mutation

Below we attach the associated code:

```
def mutation(self, chromosome):
  #Swap mutation
  gene1 = random.randint(0, len(chromosome)-1)
  gene2 = random.randint(0, len(chromosome)-1)
  while gene2 == gene1:
    gene2 = random.randint(0, len(chromosome)-1)

  chromosome[gene1], chromosome[gene2] = chromosome[gene2],
  ↪  chromosome[gene1]
  #normalization
  for i in range(len(chromosome)):
    chromosome[i] = chromosome[i]/sum(chromosome)
  return chromosome
```

Crossover:

we implemented the One point crossover with probability 0.7.

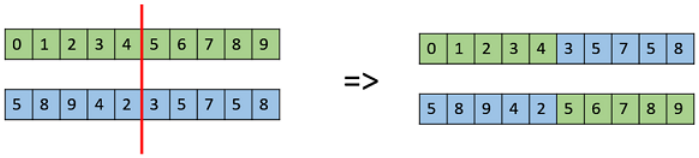


Figure 6.4: One-Point Crossover

Below we attach the associated code:

```
def crossover(self, parent1, parent2):
    #One-point crossover
    index = random.randint(1, len(parent1)-1)
    children1 = []
    children2 = []
    for i in range(len(parent1)):
      if i ¡ index:
        children1.append(parent1[i])
        children2.append(parent2[i])
      else:
        children1.append(parent2[i])
        children2.append(parent1[i])
    #normalization
    for i in range(len(children1)):
      children1[i] = children1[i]/sum(children1)
      children2[i] = children2[i]/sum(children2)
    return children1, children2
```

Parent Selection:
we implemented a Tournament Selection choosing five chromosome each time.

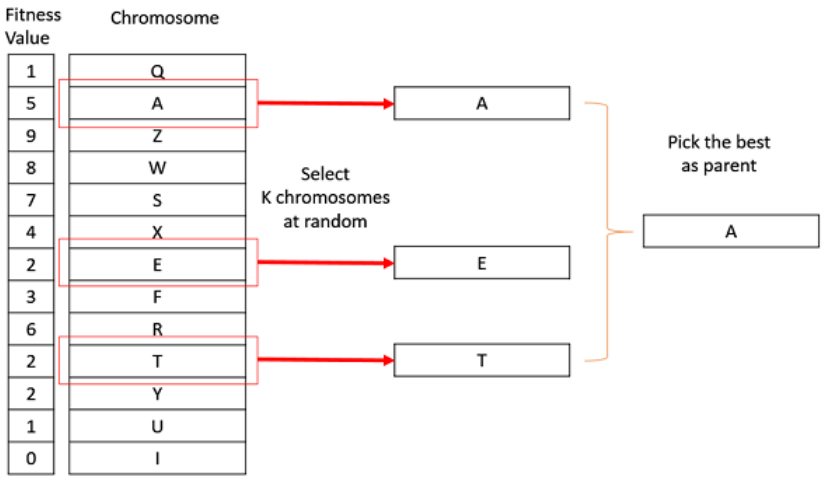


Figure 6.5: Tournament Selection for choosing parents

This method randomly samples K chromosomes and selects as a candidate parent for the off-springs generations the one with the highest fitness value among the ones sampled. Below we attach the associated code:

```
def turnamentselection(self, k):
    selectedindexes = random.sample(range(self.popsize), k)
    selectedparents = [self.population[index] for index in
    ↪  selectedindexes]
    selectedscores = [self.scores[index] for index in
    ↪  selectedindexes]

    bestparent = None
    bestscore = -1
    for i in range(k):
      if selectedscores[i] ¿ bestscore:
        bestscore = selectedscores[i]
        bestparent = selectedparents[i]

    return bestparent
```

We decided to run 15 iteration of the genetic algorithm to obtain a even more fine grained result. Each iteration has 10 generations with a population size of 100.

```
bestchromosomeovergeneticiter = []
bestfitovergeneticiter = 0
for i in range(15):
  print("Genetic iteration number " + str(i))
  bestchromosome, bestfit = ga.executeGA(ensemble)
  if bestfit ¿ bestfitovergeneticiter:
    bestfitovergeneticiter = bestfit
    bestchromosomeovergeneticiter = bestchromosome
  print("----------------------------------------")
  print("“n")
ensemble.weightedensembleprediction(bestchromosomeovergeneticiter)
```

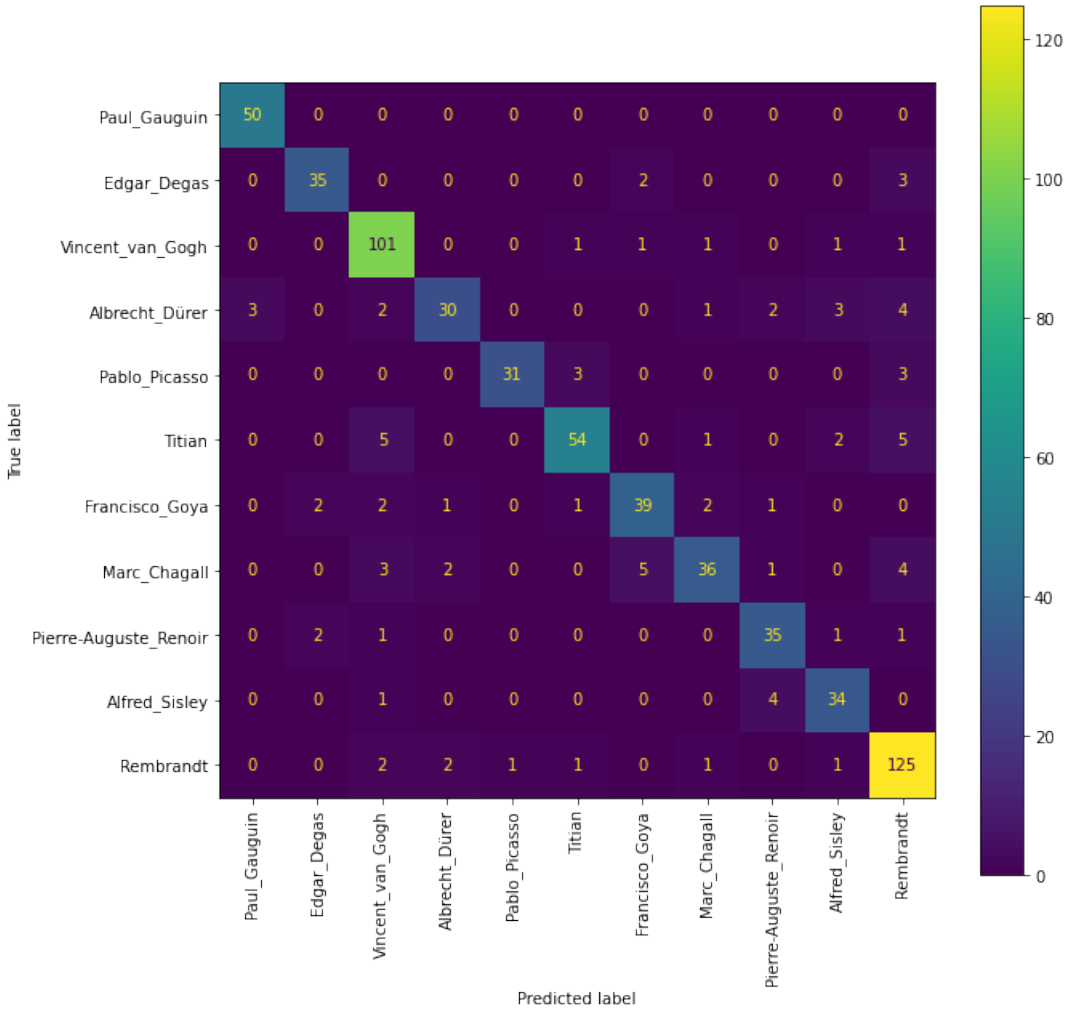


Figure 6.6: Ensemble with GA weights, Confusion Matrix

Below we attach the output of our GA algorithm over 15 iterations, applied for investigating the best combination of the weights for the ensemble's weighted average voting:

```
Genetic iteration number 0
Generation 0: Best Chromosome = [0.28671234881964286, 0.26973718119258994,
    0.40314503946580416, 0.14242062097537037] Best Fitness =
    0.8597560975609756
Generation 1: Best Chromosome = [0.2917229598177559, 0.30611014014855714,
    0.24146639631143102, 0.14507252981009294] Best Fitness =
    0.864329268292683
Generation 2: Best Chromosome = [0.22368530645284426, 0.3295203094989851,
    0.2083477191456502, 0.2544877084257318] Best Fitness =
    0.8658536585365854
Generation 3 did not find a better solution
Generation 4 did not find a better solution
Generation 5 did not find a better solution
Generation 6 did not find a better solution
Generation 7: Best Chromosome = [0.3051782747123296, 0.3158760806410364,
    0.14830577243851845, 0.2227892762894832] Best Fitness =
```

```
   0.8673780487804879
Generation 8 did not find a better solution
Generation 9 did not find a better solution
Best overall results: Best Chromosome = [0.3051782747123296,
   0.3158760806410364, 0.14830577243851845, 0.2227892762894832] Best
   Fitness = 0.8673780487804879
--------------------------------------------------------------------------------

.
.
.
.

Genetic iteration number 14
Generation 0: Best Chromosome = [0.24041719806568337, 0.2779163305011767,
   0.429093929182208, 0.432120506778939] Best Fitness = 0.8628048780487805
Generation 1: Best Chromosome = [0.18292082170204654, 0.2563734614800701,
   0.34610423949311586, 0.33902365563678627] Best Fitness =
   0.8658536585365854
Generation 2 did not find a better solution
Generation 3 did not find a better solution
Generation 4 did not find a better solution
Generation 5 did not find a better solution
Generation 6 did not find a better solution
Generation 7: Best Chromosome = [0.23232727425629365, 0.32540651536026205,
   0.19836847425787268, 0.234288832953927] Best Fitness =
   0.8673780487804879
Generation 8 did not find a better solution
Generation 9 did not find a better solution
Best overall results: Best Chromosome = [0.23232727425629365,
   0.32540651536026205, 0.19836847425787268, 0.234288832953927] Best
   Fitness = 0.8673780487804879
--------------------------------------------------------------------------------
```

Listing 6.2: output of GA algorithm over 15 iterations

Here are attached the results obtained over the test set:

```
Accuracy on test set: 0.8689024390243902
                      precision    recall  f1-score   support

         PaulGauguin     0.9434    1.0000    0.9709        50
          EdgarDegas     0.8974    0.8750    0.8861        40
     VincentvanGogh     0.8632    0.9528    0.9058       106
       AlbrechtDurer     0.8571    0.6667    0.7500        45
        PabloPicasso     0.9688    0.8378    0.8986        37
              Titian     0.9000    0.8060    0.8504        67
       FranciscoGoya     0.8298    0.8125    0.8211        48
         MarcChagall     0.8571    0.7059    0.7742        51
Pierre-AugusteRenoir     0.8140    0.8750    0.8434        40
        AlfredSisley     0.8095    0.8718    0.8395        39
           Rembrandt     0.8562    0.9398    0.8961       133

            accuracy                         0.8689       656
           macro avg     0.8724    0.8494    0.8578       656
```

```
18          weighted avg     0.8702    0.8689    0.8667       656
19
20 Best Chromosome = [0.33499967323064334, 0.32215826949236503,
   0.11083341523402779, 0.23186808567210185]
```

Listing 6.3: Classification result over the test set

As we can observe we reached an accuracy score of nearly 87% (+0.02) and a weighted-F1 score of 86%, the best one so far. It outperforms the classical ensemble method thanks to the combination of different weights for different classifiers in the average voting. The weights are assigned accordingly to their performances on the test set, specifically, the highest weight is associated to the most precise model (ResNet50) while the smallest weight is associate to the least precise model.// It can be consider a satisfactory result, also comparing it to the state of the art results for multi-class classification tasks over the same complex argument.

Chapter 7

Explainability

When working with neural networks, it's important to understand how they classify images. The technique used is to visualize the activations of intermediate layers and display heatmaps.

7.1 Intermediate Activations

We take as example the following image:

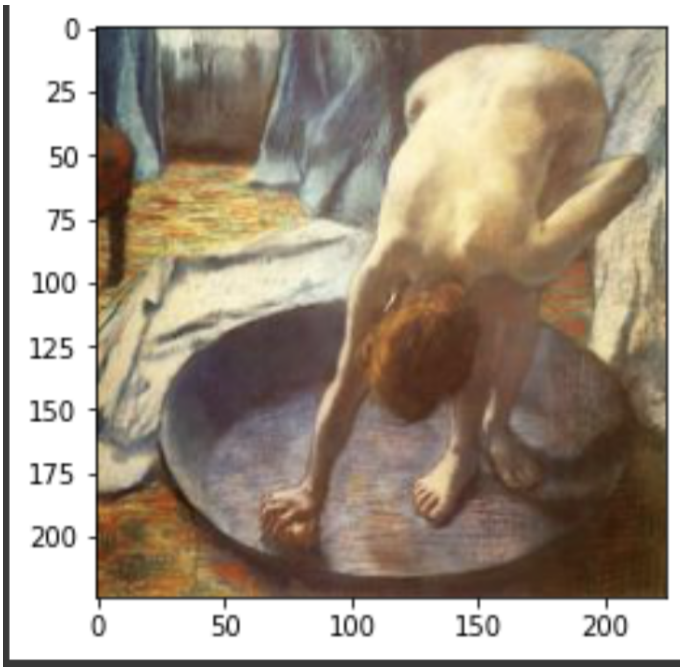
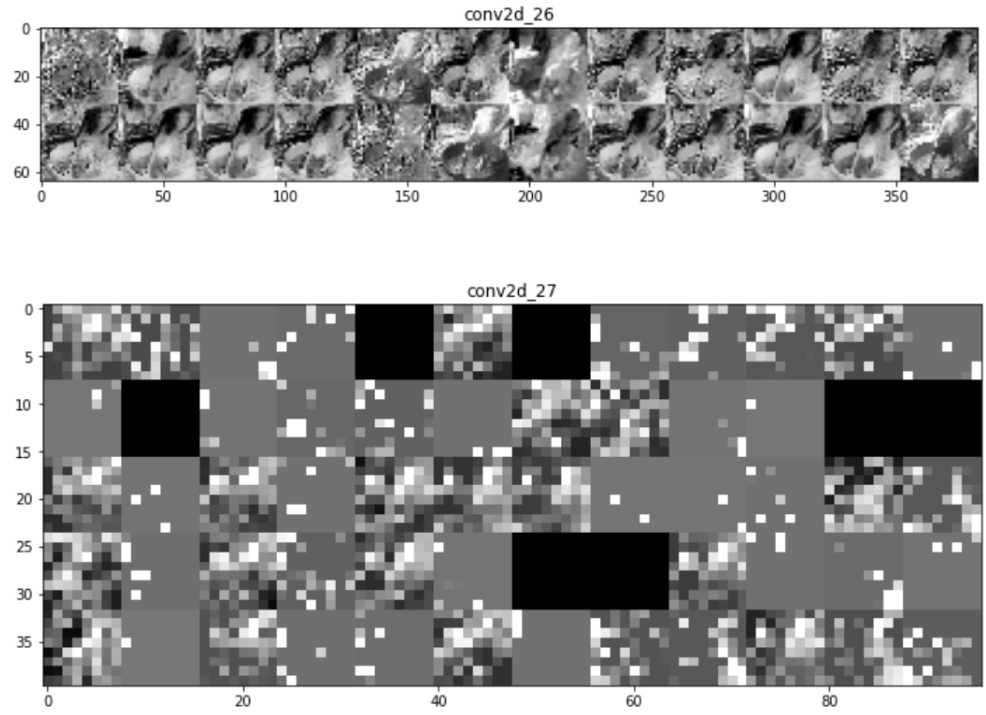


Figure 7.1: Edgar Degass

7.1.1 From Scratch Network

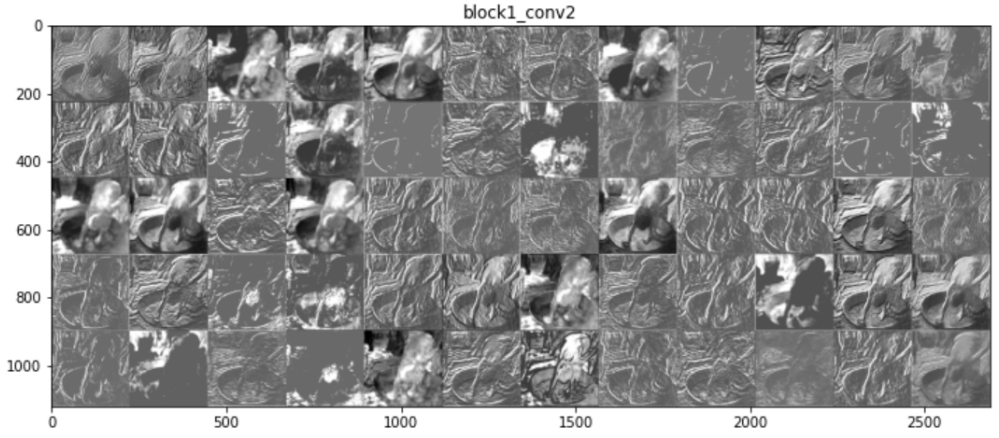
Let's begin by looking at the "from scratch" network we created. In the first convolutional layer, the network focuses on the edges present in the image to be classified. However, in a

later layer, the activations become more abstract and less visibly interpretable. We also notice that some feature maps are not activated, as they do not provide relevant information for the image. Mathematically, this is due to the use of ReLU, which sets the output to 0 for negative values.



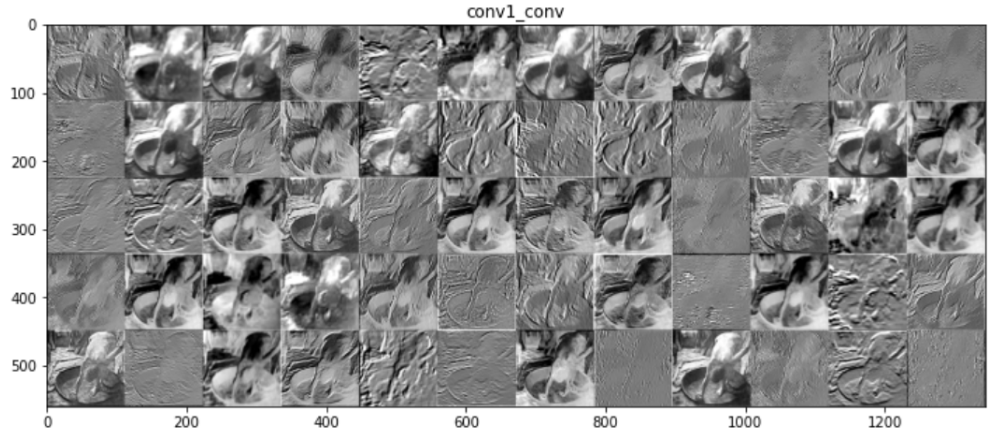
7.1.2 VGG16

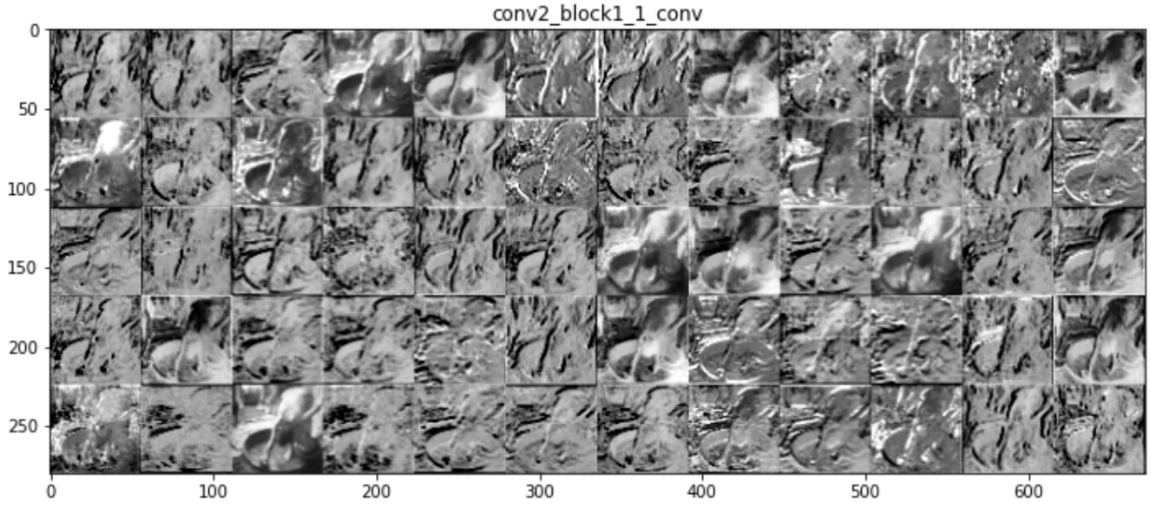
Regarding VGG16, we can see that in one of the first convolutional layers, the feature maps focus on edges, lines, and shapes. We also notice the ”prominence” that some feature maps have. Comparing the results obtained with the ”from scratch” network and VGG16, it’s clear that the latter has more ability to recognize different characteristics of the given image, thanks to its more complex architecture and the fact that it was trained on a much larger number of images than the ”from scratch” network.



7.1.3 Resnet50

For Resnet50, the situation is quite similar to VGG16. In the second image, which is from a subsequent convolutional layer, the focus is more on certain nuances and edges.





7.2 Heatmap

Now we are using a technique that will help us understand which parts of the given image contribute to the final classification decision. We take a work of art by the great artist Pablo Picasso and see what the different networks focus on.

7.2.1 From Scratch

It is evident for the from scratch network that we have a high granularity of the features map of the last layer. The heatmap is interpreted as follows: the lighter the color, the greater the intensity of the activation of that part of the feature map.

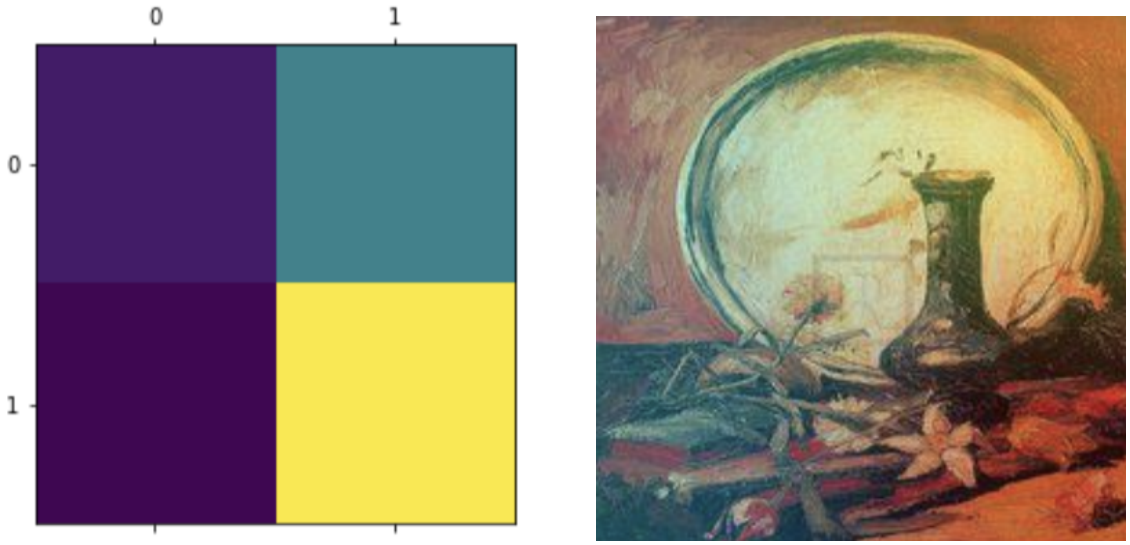


Figure 7.2: Heatmap and Gradcam

7.2.2 VGG16

Here we can see that the granularity is higher, in fact we have feature maps that are 7x7 instead of 2x2 as in the case of the from scratch network. We see that the network's attention for the classification of this artwork is in the central part of the image.

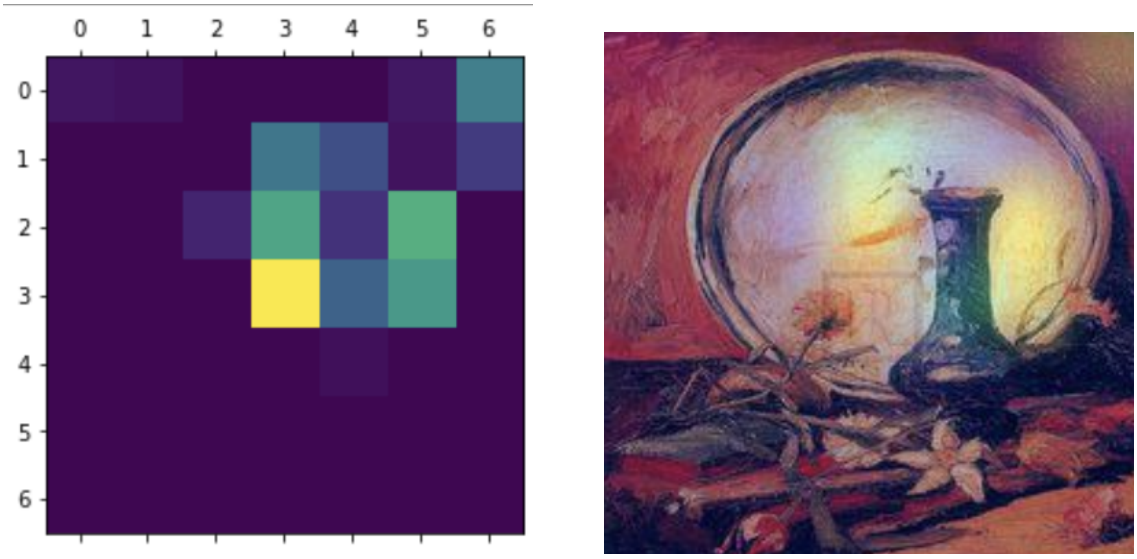


Figure 7.3: Heatmap and Gradcam

7.2.3 Resnet50

The last layer focuses on different details of the image compared to what happens with VGG16. In fact, we see that in the case of Resnet50, a large part of the image is taken into consideration, while with the previously seen network, there was only a great attention in the center of the image.

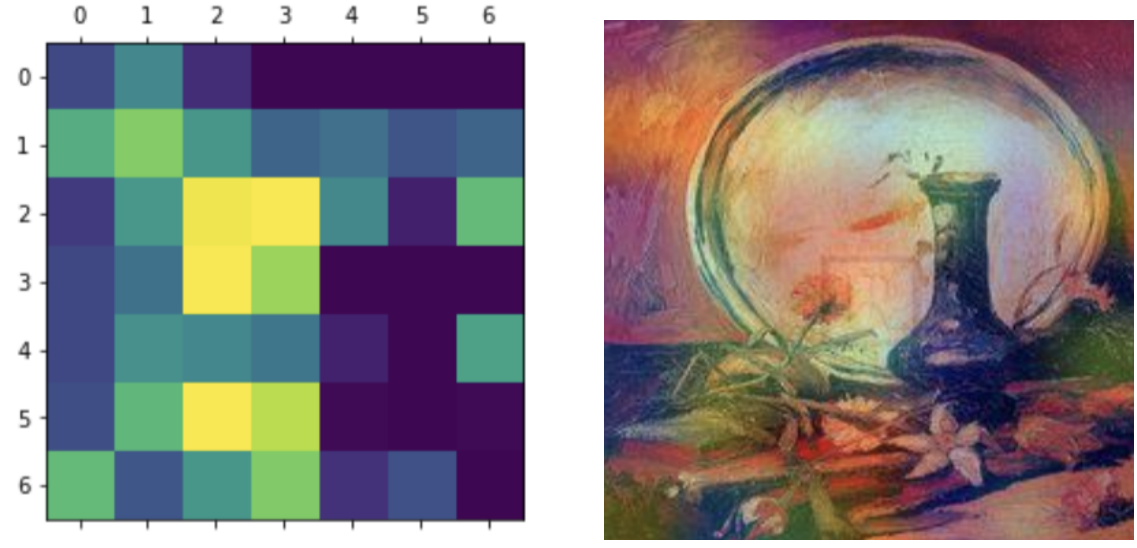


Figure 7.4: Heatmap and Gradcam

Chapter 8

Conclusions

To sum up we obtained reasonable results despite of the complexity of the classification task, specifically the high number of classes and the difficulties that come from the similarities of some art styles. We started out collecting some information regarding the state of the art in this particular field to have an idea of what was already been tested and the most effective ways to tackle this problem.

Fist of all we implemented out from scratch model with a trial and error approach, evaluating for each trial the validation and test accuracy and loss. The results obtained with our network was not very promising, reaching the highest value of accuracy of 67%. Next, we managed to get better performances with three pretrained models (VGG16, ResNet50 and InceptionV3). The best results were obtained thanks to ResNet50, that manages to classify with greater accuracy (82%) w.r.t. our network, thanks to the already-set weights and pre-made feature maps.

Much more satisfactory results have been obtained thanks to the **Ensemble**, that merges our best from scratch network and the best pretrained models. We managed to achieve and accuracy of 86% on test set, we went further exploring the application of a simple Genetic Algorithm applied to the average voting paradigm of the aforementioned ensamble model. Specifically, we exploited the GA to choose among the best combinations of four weights to assign to each different model. With this enhancement we managed to reach 87% accuracy on test set.

To conclude, we were able to achieve comparable results with the ones mentioned in section 1.1, also considered the constrained dimension of our dataset (89% accuracy on 23.817 images for about 90 GB of data, VS our 87% accuracy on 3004 images for 1 GB of data).

References

[1] Lombardi and Thomas Edward. The classification of style in fine-art painting. 2005.

[2] Jonathan Jou and Sandeep Agrawal. Artist identification for renaissance paintings. 2011.

[3] Thomas Mensink and Jan van Gemert. The rijksmuseum challenge: Museum-centered visual recognition. In *Proceedings of International Conference on Multimedia Retrieval*, ICMR '14, page 451–454, New York, NY, USA, 2014. Association for Computing Machinery.

[4] Babak Saleh and Ahmed M. Elgammal. Large-scale classification of fine-art paintings: Learning the right metric on the right feature. *CoRR*, abs/1505.00855, 2015.

[5] Kaggle Competition. https://www.kaggle.com/c/painter-by-numbers/data.

[6] Nitin Viswanathan and Stanford. Artist identification with convolutional neural networks. 2017.

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

[8] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *Journal of Machine Learning Research*, 15(56):1929–1958, 2014.

[9] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[10] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition, 2014.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[12] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *CoRR*, abs/1512.00567, 2015.