UNIVERSITÀ DI PISA

Artificial Intelligence and Data Engineering

Computational Intelligence and Deep Learning

Artist Identification with Convolutional Neural Networks

Project Documentation

TEAM MEMBERS:
Edoardo Fazzari
Mirco Ramo

Academic Year: 2020/2021

Contents

1 Introduction **3**
- 1.1 State of the Art . . . 3
- 1.2 Dataset . . . 4

2 General Information Useful for Training **7**
- 2.1 Data Augmentation . . . 7
- 2.2 Regularization . . . 7
- 2.3 Dropout . . . 8
- 2.4 Activation Functions . . . 8
- 2.5 Optimizers . . . 8
- 2.6 Genetic Algorithms . . . 8
  - 2.6.1 Elitism . . . 9

3 CNN from Scratch **10**
- 3.1 Standard CNN . . . 10
  - 3.1.1 Simple model . . . 10
  - 3.1.2 Deeper network and dropout . . . 11

4 Pre-Trained Models **12**
- 4.1 VGG16 . . . 12
  - 4.1.1 Test 1: Classical VGG16 (Feature Extraction) . . . 12
  - 4.1.2 Test 2: Adding Dropout to Test 1 . . . 13
  - 4.1.3 Test 3: Finetuning One Convolutional Layer . . . 14
  - 4.1.4 Test 4: test 3 with dropout and different optimizer . . . 15
  - 4.1.5 Test 5: Finetuning Two Convolutional Layers . . . 15
  - 4.1.6 Test 6: Finetuning One Convolutional Layer and Weights Regularization 16
  - 4.1.7 Test 7: Finetuning Two Convolutional Layers and Weights Regularization 17
  - 4.1.8 Test 8: Genetic Algorithm for Hyper-parameters and Architecture Optimization . . . 17
- 4.2 ResNet50V2 . . . 19
  - 4.2.1 Test 1: Classical ResNet50V2 (Feature Extraction) . . . 19
  - 4.2.2 Test 2: Finetuning 1 block . . . 20
  - 4.2.3 Test 3: Finetuning 2 blocks . . . 20
  - 4.2.4 Test 4: Finetuning with One Block and Adding Two Dense layers . . . 21
  - 4.2.5 Test 5: Finetuning with Two Blocks and Adding Two Dense layers . . . 21
- 4.3 ResNet101V2 . . . 22
  - 4.3.1 Test 1: Classical ResNet101V2 . . . 22
  - 4.3.2 Test 2: Finetuning One Sub-Block . . . 23
  - 4.3.3 Test 3: Finetuning the Entire Block 5 . . . 23
  - 4.3.4 Test 4: Finetuning Half Block 4 . . . 24
  - 4.3.5 Test 5: Test 4 and Dropout . . . 24
  - 4.3.6 Test 6: Adding Dense Layers to Test 4 . . . 25
  - 4.3.7 Test 7: Going Deeper and Deeper . . . 26
- 4.4 InceptionV3 . . . 27
  - 4.4.1 Test 1: Classical InceptionV3 . . . 27
  - 4.4.2 Test 2: Finetuning 1 Block . . . 28
  - 4.4.3 Test 3: Finetuning 2 Blocks . . . 28
  - 4.4.4 Test 4: Finetuning 3 Blocks . . . 29

4.4.5 Test 5: Finetuning 3 Blocks with Dropout 30
4.4.6 Test 6: Finetuning 3 Blocks and ExponentialDecay Learning Rate 31

5 Ensemble Network **32**
5.1 Our approach 32
5.1.1 The Ensemble Classese 32
5.1.2 Genetic Algorithm Workflow 35
5.1.3 GA Results for VGG16 Ensemble 37
5.1.4 GA Results for ResNet Ensemble 38
5.1.5 GA Results for ResNet Inception 39

6 Conclusion **40**

1 — Introduction

Artist identification is traditionally performed by *art historians* and *curators* who have expertise and familiarity with different artists and styles of art. This is a complex and interesting problem for computers because identifying an artist does not just require object or face detection; artists can paint a wide variety of objects and scenes. Additionally, many artists from the same time period will have similar styles, and some such as **Pablo Picasso** (see figure 1) have painted in multiple styles and changed their style over time.

Figure 1: Both of these paintings were created by Pablo Picasso, but they have vastly different styles and content.

The aim of this project is to use Convolutional Neural Networks for the identification of an artist given a painting. In particular, the CNN networks will be modeled using multiple techniques: from scratch; via pretrained network; networks based on comparison with baseline input and using an ensemble network made of by the best classifiers found.

1.1 State of the Art

As mentioned, artist identification has primarily been tackled by humans. An example of that is the Artsy's Art Genome Project [1], which is led by experts who manually classify art. This strategy is not very scalable even if it is highly precise in the classification (the site is a marketplace of fine-arts for collects, you can find Pisarro, Bansky and other famous artists).

Most prior attempts to apply machine learning to this problem have been feature-based, aiming to identify what qualities most effectively distinguish artists and styles. Many generic image features have been used, including scale-invariant feature transforms (SIFT), histograms of oriented gradients (HOG), and more, but with the focus on *discriminating different style* in Fine-Art Painting[2].

The first time the problem of artist identification was really tackled was with J. Jou and S. Agrawal[3] in 2011, they applied several multi-class classification techniques like Naïve Bayes, Linear Discriminant Analysis, Logistic Regression, K-Means and SVMs and achieve a maximum classification accuracy of 65% for an unknown painting across 5 artists. Later on, the problem of identifying artists was retackled by the *Rijksmuseum Challenge*[4]. The objective of the challenge was to predict the artist, type, material and creation year (each of them was a different challenge) of the 112,039 photographic [5] (containing different viewpoints of an artwork, and different types of them: sculptures, paintings, saucers, etc.) reproductions of the artworks exhibited in

[1] https://www.artsy.net/categories

[2] T. E. Lombardi. The classification of style in fine-art paint- ing. ETD Collection for Pace University, 2005

[3] J. Jou and S. Agrawal. Artist identification for renaissance paintings.

[4] T. Mensink and J. van Gemert. The rijksmuseum challenge: Museum-centered visual recognition. 2014

[5] The dataset contains 6,629 artists in total, with high variation in the number of pieces per artist. For example, Rembrandt has 1,384 pieces, and Vermeer has only 4. There are 350 artists with more than 50 pieces, 180 artists have around 100, and 90 artists have 200 pieces.

the Rijksmuseum in Amsterdam (the Netherlands). For the artist classification challenge, the paper said they reached a test accuracy of about 60%. The year later, Saleh and Elgammal's paper[6] was the first attempt to identify artists with a large and varied dataset, but still using generic features. The collection they used has images of 81,449 fine-art paintings from 1,119 artists ranging from fifteen centuries to contemporary artists, reaching an accuracy of 59%[7].

More recent attempts are related to the *Painter by Numbers*, a **Playground Prediction Competition** by *Kaggle*[8]. This competition used a pairwise comparison scheme: participants had to create an algorithm which needs to examine two images and predict whether the two images are by the same artist or not. Thus, it is not our same objective, however it can be consider the first application of Deep Learning to the problem. The real deal was taken by Nitin Viswanathan[9] in 2017. Viswanathan, using the same dataset of the mentioned *Kaggle Challenge*, proposed the use of ResNet with transfer learning (he first held the weights of the base ResNet constant and updated only the fully-connected layer for a few epochs). This trained network reached a train accuracy of 0.973 and a test accuracy of 0.898.

1.2 Dataset

Unfortunately, the dataset provided by the *Kaggle Challange* is huge thus unfeasible to be used in Colab: in fact it is about 60GB unbearable on the free version of Colab, which provides only about 30GB of disk. Stated that, we decided to use a different dataset[10] with only 2GB of data and about 8k unique images.
The data downloaded from Kaggle has the following directories and csv file:

```
/
├── images
│   └── images
│       ├── Albrecht_Durer
│       ├── Alfred_Sisley
│       ├── Amedeo_Modigliani
│       ├── Andrei_Rublev
│       ├── Andy_Warhol
│       ├── Camille_Pissarro
│       ├── Caravaggio
│       ├── Claude_Monet
│       ├── Diego_Rivera
│       ├── Diego_Velazquez
│       ├── Edgar_Degas
│       ├── Edouard_Manet
│       ├── Edvard_Munch
│       └── an many others (total of 50 different artists)
├── resized
└── artists.csv
```

The *resized* directory is not useful for our studies, hence we deleted it to save space on the disk. On the other hand, we first use the *csv* file to select only the artists with at least 200 pieces, 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. Even done that, the dataset was still unbalanced, e.g. Van Gogh's paintings are 877 against the 239 of Chagall's,

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

[7] In the paper they tried to use also CNN, but reaching only an accuracy of 33.62%

[8] https://www.kaggle.com/c/painter-by-numbers/data

[9] Nitin Viswanathan, Artist Identification with Convolutional Neural Networks

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

thus we consider to compute **class weights** in order to use them in the *fit function*:

$$\text{class\_weights} = \frac{\text{Total number of paintings considered}}{\text{Number of artists considered} \cdot \text{Number of paintings per author}}$$

Then, we modified the structure of the *images/images* directory in order to create two directories, **train** and **test**, containing 90% and 10% of the images from each different artist's directory respectively (considering only the artists with at least 200 paintings). The newly created directories have the same structured of *images/images*. This was done in *python* in this way:

```
import os
import numpy as np
import shutil

rootdir= '/content/images/images' #path of the original folder
classes = os.listdir(rootdir)

for i, c in enumerate(classes, start=1):
  if c not in artists_top_name.tolist():
    shutil.rmtree(rootdir + '/' + c)
    continue
  if not os.path.exists(rootdir + '/train/' + c):
    os.makedirs(rootdir + '/train/' + c)
  if not os.path.exists(rootdir + '/test/' + c):
    os.makedirs(rootdir + '/test/' + c)

  source = os.path.join(rootdir, c)
  allFileNames = os.listdir(source)

  np.random.shuffle(allFileNames)

  test_ratio = 0.10
  train_FileNames, test_FileNames = np.split(np.array(allFileNames),
                                                        [int(len(allFileNames)*
                                            (1 - test_ratio))])

  train_FileNames = [source+'/'+ name for name in train_FileNames.tolist()]
  test_FileNames = [source+'/' + name for name in test_FileNames.tolist()]

  for name in train_FileNames:
    shutil.copy(name, rootdir +'/train/' + c)

  for name in test_FileNames:
    shutil.copy(name, rootdir +'/test/' + c)
```

After that we created the train/validation/test-sets using the *image_dataset_from_directory* function provided by **Keras** in the following way:

```
import tensorflow as tf

training_images = tf.keras.preprocessing.image_dataset_from_directory(
    TRAIN_DIR, labels='inferred', label_mode='categorical',
    class_names=None, color_mode='rgb', batch_size=BATCH_SIZE,
    image_size=(IMAGE_HEIGHT,  IMAGE_WIDTH), shuffle=True, seed=RANDOM_SEED,
    validation_split=VALIDATION_SPLIT, subset='training',
    interpolation='bilinear', follow_links=False
)

val_images = tf.keras.preprocessing.image_dataset_from_directory(
    TRAIN_DIR, labels='inferred', label_mode='categorical',
    class_names=None, color_mode='rgb', batch_size=BATCH_SIZE,
    image_size=(IMAGE_HEIGHT, IMAGE_WIDTH), shuffle=True, seed=RANDOM_SEED,
    validation_split=VALIDATION_SPLIT, subset='validation',
    interpolation='bilinear', follow_links=False
)
```

```
18
19 test_images = tf.keras.preprocessing.image_dataset_from_directory(
20     TEST_DIR, labels='inferred', label_mode='categorical',
21     class_names=None, color_mode='rgb', batch_size=BATCH_SIZE,
22     image_size=(IMAGE_HEIGHT, IMAGE_WIDTH), shuffle=True, seed=RANDOM_SEED,
23     interpolation='bilinear', follow_links=False
24 )
```

Where *VALIDATION_SPLIT* is equal to 0.1.
Obtaining in this way:

- 3478 files for training (belonging to 11 classes).
- 386 files for validation (belonging to 11 classes).
- 438 files for testing (belonging to 11 classes).

Hence, we have a total of 4299 different pictures.

2 — General Information Useful for Training

In the following chapters we will make use of different strategies:

- Class Weights (already talked about)
- Data augmentation
- Regularization
- Dropout
- Multiple activation functions
- Multiple optimizers
- Genetic Algorithms

In order to allow a better and faster reading of the tests done, in the following paragraph the mentioned strategies are discussed.

2.1 Data Augmentation

Data augmentation takes the approach of generating more training data from existing training samples by augmenting the samples via a number of random transformations that yield believable-looking images. The goal is that, at training time, your model will never see the exact same picture twice. This helps expose the model to more aspects of the data so it can generalize better. In Keras, this can be done by adding a number of data augmentation layers at the start of your model. In our model, we included the following transformation:

```
data_augmentation = ks.Sequential(
    [
        layers.RandomFlip('horizontal'),
        layers.RandomRotation(0.1),
        layers.RandomZoom(0.2),
        layers.RandomHeight(0.1),
        layers.RandomWidth(0.1)
    ]
)
```

2.2 Regularization

Regularization techniques are a set of best practices that actively impede the model's ability to fit perfectly to the training data, with the goal of making the model perform better during validation. This is called "regularizing" the model, because it tends to make the model simpler, more "regular", its curve smoother, more "generic"; thus it is less specific to the training set and better able to generalize by more closely approximating the latent manifold of the data. A common way to mitigate overfitting is to put constraints on the complexity of a model by forcing its weights to take only small values, which makes the distribution of weight values more regular. This is called *weight regularization*, and it's done by adding to the loss function of the model a cost associated with having large weights. This cost comes in two flavors:

1. *L1 regularization*—The cost added is proportional to the absolute value of the weight coefficients (the L1 norm of the weights).

2. *L2 regularization*—The cost added is proportional to the square of the value of the weight coefficients (the L2 norm of the weights).

3. *L1_L2 regularization*—Combine L1 and L2.

2.3 Dropout

Dropout is one of the most effective and most commonly used regularization techniques for neural networks; it was developed by Geoff Hinton and his students at the University of Toronto. Dropout, applied to a layer, consists of randomly dropping out (setting to zero) a number of output features of the layer during training.

In **keras** can be set using the *layers.Dropout* function passing as parameter the *dropout rate*. We tried different values for the dropout rate during our studies, anyway for the *Pre-Trained Models* chapters it is always set to 0.5 if not otherwise specified.

2.4 Activation Functions

In the studies done in the following chapters we used three different activation functions:

- $ReLU$: $\max(0, x)$
- ELU: $\max(0.2x, x)$

They will be useful in the genetic algorithm analysis done fore the *scratch architecture* and the *VGG16*

2.5 Optimizers

An optimizer is the mechanism through which the model will update itself based on the training data it sees, so as to improve its performance. In our project we make use of:

- *RMSprop*: the gist of RMSprop is to:
  - Maintain a moving (discounted) average of the square of gradients
  - Divide the gradient by the root of this average
  - It uses plain momentum, not Nesterov momentum.
- *Adam*: stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments.

2.6 Genetic Algorithms

Genetic algorithms are a family of search algorithms inspired by the principles of evolution in nature. By imitating the process of natural selection and reproduction, genetic algorithms can produce high-quality solutions for various problems involving search, optimization, and learning. At the same time, their analogy to natural evolution allows genetic algorithms to overcome some of the hurdles that are encountered by traditional search and optimization algorithms, especially for problems with a large number of parameters and complex mathematical representations. Thus, they come in handy for optimizing our networks. In order to make use of genetic algorithms we must decide some components, which are:

- *Genotype*: the *genotype* is a collection of genes that are grouped into chromosomes.
- *Population*: at any point in time, genetic algorithms maintain a population of individuals (i.e., chromosomes)– a collection of candidate solutions for the problem at hand.

- *Fitness Function*: at each iteration of the algorithm, the individuals are evaluated using a fitness function (also called the target function). This is the function we seek to optimize or the problem we attempt to solve.
- *Selection Algorithm*: after calculating the fitness of every individual in the population, a selection process is used to determine which of the individuals in the population will get to reproduce and create the offspring that will form the next generation.
- *Crossover Algorithm*: to create a pair of new individuals, two parents are chosen from the current generation, and parts of their chromosomes are interchanged (crossed over) to create two new chromosomes representing the offspring.
- *Mutation Algorithm*: the purpose of the mutation operator is to periodically and randomly refresh the population, introduce new patterns into the chromosomes, and encourage search in uncharted areas of the solution space.
- *Elitism*: described in the following paragraph.

All of these components are implemented using the python library **deap**[11].

2.6.1 Elitism

While the average fitness of the genetic algorithm population generally increases as generations go by, it is possible at any point that the best individual(s) of the current generation will be lost. This is due to the selection, crossover, and mutation operators altering the individuals in the process of creating the next generation. In many cases, the loss is temporary as these individuals (or better individuals) will be re-introduced into the population in a future generation.

However, if we want to guarantee that the best individual(s) always make it to the next generation, we can apply the optional elitism strategy. This means that the top n individuals (n being a small, predefined parameter) are duplicated into the next generation before we fill the rest of the available spots with offspring that are created using selection, crossover, and mutation. The elite individuals that were duplicated are still eligible for the selection process so they can still be used as the parents of new individuals.

Elitism is made possible in our code thanks to the function *eaSimpleWithElitism*, which is a modification of the function *eaSimple* present in the **Deap** framework.

[11] https://deap.readthedocs.io/en/master/

3 — CNN from Scratch

This chapter shows the results of the training of several custom architecture, that have been defined in order to solve the classification task. Starting from a very simple model, we start to analyze how to improve it and what modifications to apply in order to improve performance, taking into account mainly the accuracy on the validation test, but also considering other metrics like training time or number of parameters. The overall strategy is the following:

- test of different custom architectures defined from scratch
- analysis of the level of fitting, try of different techniques to fight possible underfitting/overfitting
- Experiment with the addition of Batch Normalization
- hyperparameters optimization on the best model so far using a Genetic Algorithm(see Section 2.6)

The objective of the presented procedure is not the total exploration and exploitation of the search space, but it aims at finding good results in a reasonable time exploiting an ad-hoc heuristic search. The tested models are the following:

3.1 Standard CNN

3.1.1 Simple model

The first experiment has been conducted using a customized standard CNN that exploits Convolutional Layers and max Pooling to process input images. To start, we defined a very simple model, whose structure is reported in the image 2. This network is mainly a starting point of our trial-and-error approach and will give us a first approximation of what is going to be our prediction power on the considered task.

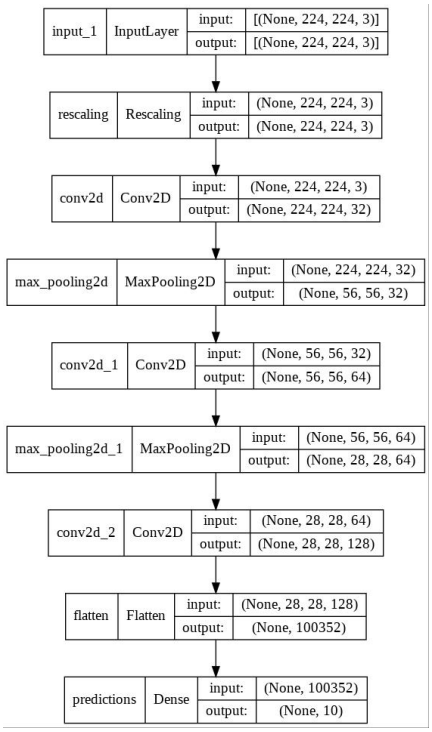


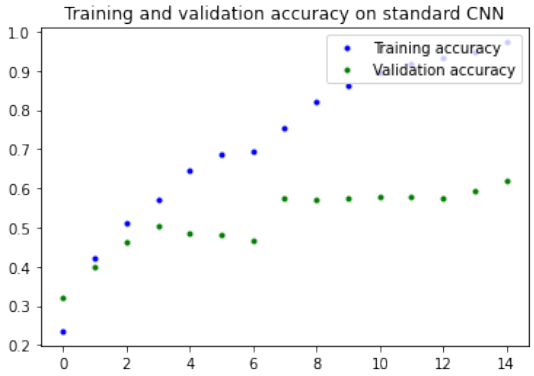
Figure 2: Customized Standard CNN Architecture

This model is trained using these default hyperparameters:

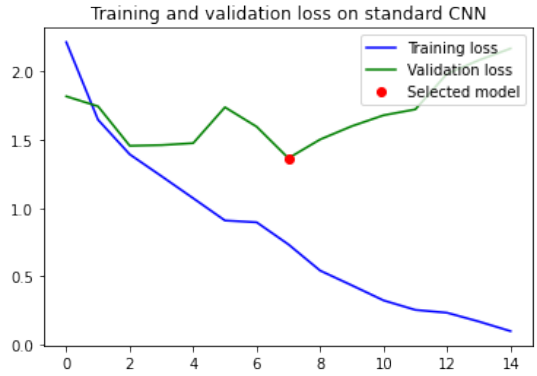
- optimizer: *ADAM*
- dropout rate :0.0
- learning rate: optimizer's default
- batch size: 128
- learning rate decay: none

In particular, we set a large value for batch size both because our main goal is to maximize the accuracy and also (as presented in the Introduction) to face off with the great variability of paintings with very dissimilar style but belonging to the same author. In this way, we increase the probability of a batch to be "complete", thus being representative of this variability.
The results obtained are the following:

StandardCNN				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
15	0.5730	0.786	1,3656	0.6542



(a) Standard CNN Accuracy



(b) Standard CNN Loss

The network reaches an accuracy value equal to 57.3% in just 8 epochs, then it overfits very quickly. It can be caused by mainly 3 factors:

- Data available is insufficient, so the network loses the ability to generalize
- Lack of regularization techniques, such as Dropout or L1/L2 regularizations
- **The spatial extent of the feature map of the last layer is still quite large, so the fully connected layers have too many parameters operating in a quite shallow representation.**

3.1.2 Deeper network and dropout

4 — Pre-Trained Models

This section describes the results obtained using different pre-trained architecture and strategies[12]. The pre-trained networks here tested are:

- VGG16
- ResNet50V2
- ResNet101V2
- InceptionV3

4.1 VGG16

VGG164 is a convolutional neural network model proposed by Simonyan et al., with several 3x3 convolutional layers in cascade occasionally interleaved with 2x2 max-pooling layers forming the so called *blocks*. Developed for the ILSVRC2014 challenge, it was able to achieve a top-5 accuracy of 92.7 on ImageNet.

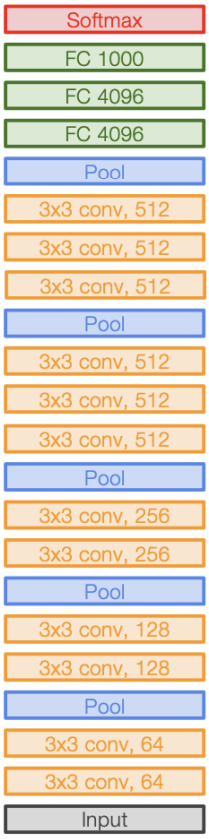


Figure 4: VGG16 Architecture

4.1.1 Test 1: Classical VGG16 (Feature Extraction)

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

[12]The data augmentation strategy is always used since we have very little data

base is left as it is, and the fully-connected block is replaced by the a shrunk version with only 256 neurons per layer, followed by our prediction layer made of 11 neurons5.

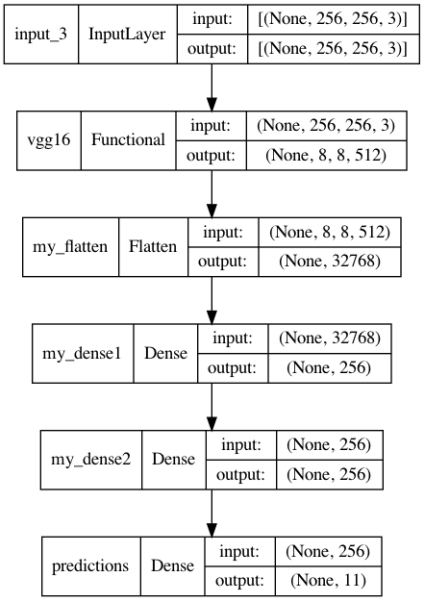
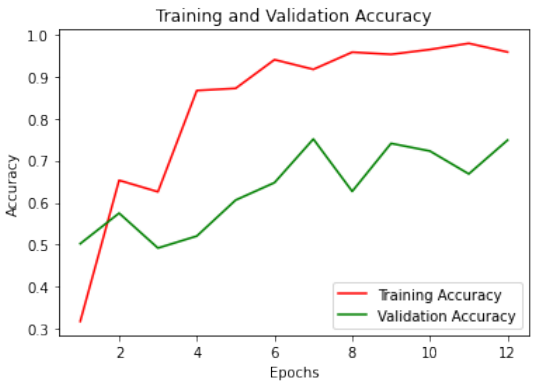


Figure 5: Our Feature Extraction Network

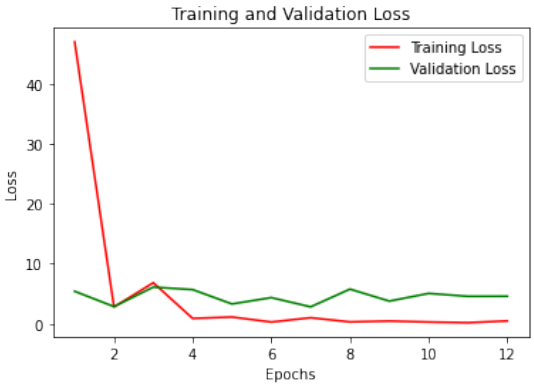
The result obtained, using RMSprop as optimizer, are:

Feature Extraction				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
12	0.7409	0.7057	5.1959	5.7

The network begin to overfit very fast, hence some regularization methods are needed.



(a) Simple VGG16 Feature Extraction Accuracy



(b) Simple VGG16 Feature Extraction Loss

4.1.2 Test 2: Adding Dropout to Test 1

We have two possible positions to use the dropout layer in our network and they are after each 256-dense layer, but we decided to use just one layer at the end of the second 256-Dense layer (*my_dense1*) as shown in Figure7. We didn't use a dropout layer between the two 256-dense layers, since this type of architecture led to worst performance, this mainly because we would have less units to fully train our topic-specific network.

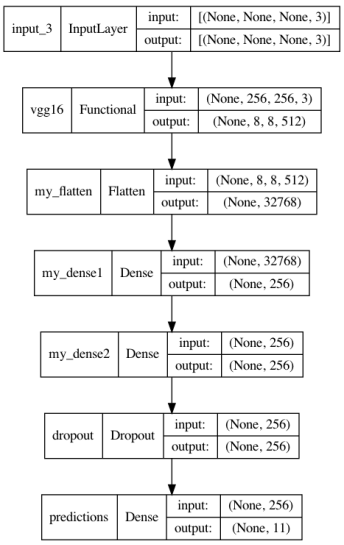
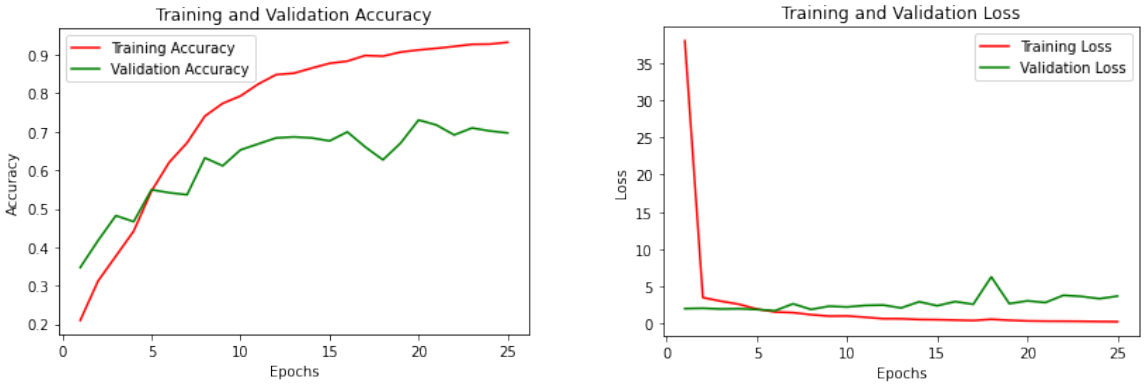


Figure 7: Our Feature Extraction Network + Dropout

The result obtained, using RMSprop as optimizer, are:

Feature Extraction w/ dropout				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
25	0.7306	0.7195	3.0616	3.0035



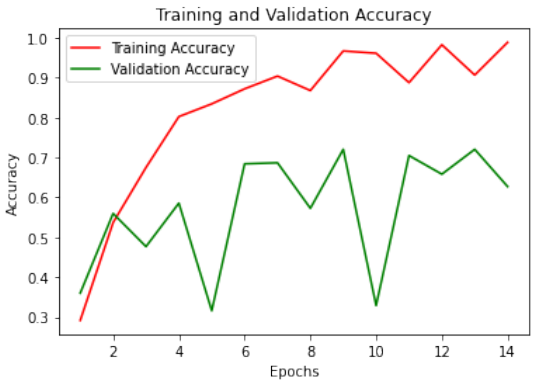
(a) Test 2 Accuracy **(b)** Test 2 Loss

As expected, dropout mitigated the magnitude of overfitting, however our network perform slightly worst (now the validation accuracy is 0.73 and before was 0.74) than without the dropout layer.

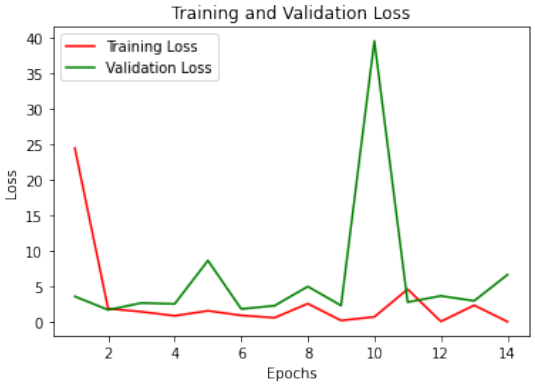
4.1.3 Test 3: Finetuning One Convolutional Layer

Using the model defined in test 1, the 3rd Conv2D layer in the 5th block is un-fronzen and the network is trained. The result obtained using RMSprop as optimizer are the following:

Finetuning one convolutional layer				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
28	0.7202	0.6253	2.3687	8.1807



(a) Test 3 Accuracy



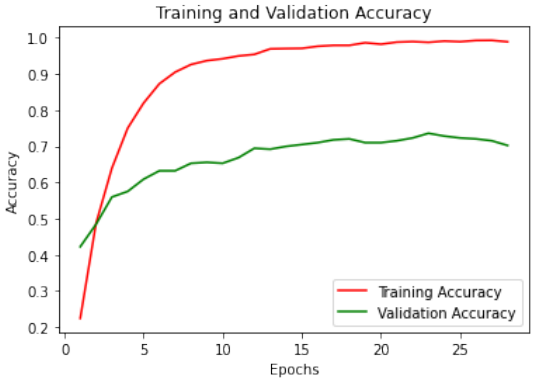
(b) Test 3 Loss

Again the performance are worst than in test one, but looking at the graphs it can be seen that not only our network overfitted very fast, but it forms also few fang-shaped changes in direction.

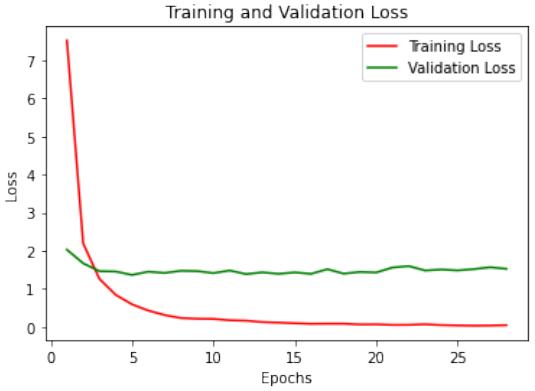
4.1.4 Test 4: test 3 with dropout and different optimizer

To overcome the previous problems, the overfitting and the strange shape behavior, in this test we opt to use **Adam** as an optimizer, changing its default learning rate (i.e., 0.001) to 0.0001 in order to slowly learn and hoping to have a smoother accuracy and loss functions.

Finetuning one conv layer w/ dropout and Adam				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
28	0.7358	0.7218	0.9871	1.1792



(a) Test 4 Accuracy



(b) Test 4 Loss

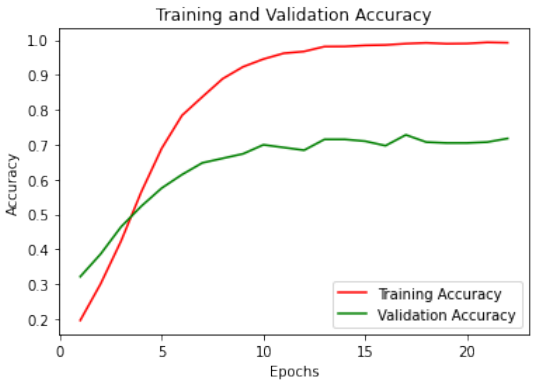
Adding dropout and a different optimizer with a little learning rate, we finally obtained what we were aiming. Anyway, the result is now comparable to the test 1, however now we our using a more complex network which is not good if the results are the same.

However we can still do something, the training accuracy increases very rapidly even if dropout is applied. To decrease this effect in test 6 we use weight regularization techniques.

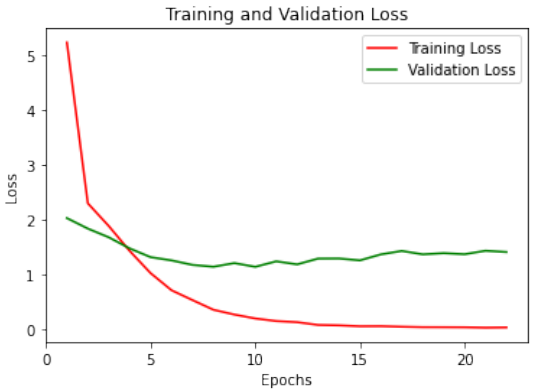
4.1.5 Test 5: Finetuning Two Convolutional Layers

This test is basically test 4, but finetuning the last two convolutional layers of VGG16.

Finetuning two convolutional layers				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
22	0.7280	0.7379	1.4253	1.2589



(a) Test 5 Accuracy



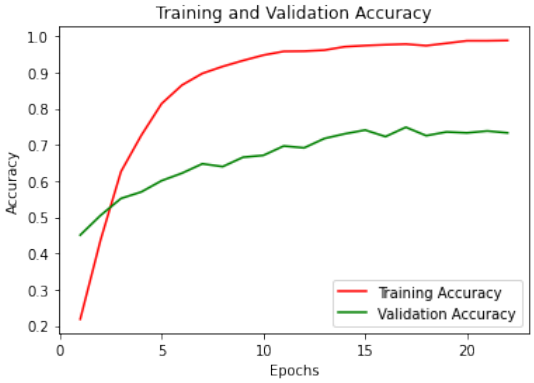
(b) Test 5 Loss

Following the approach used in the previous test we obtained also here more smoothed graphs, but the performance is little lower than before. The problem here is that we are propagating to the second layer in block 5 gradients that are not really improvements of the ones set by the *imagenet* default configuration.

4.1.6 Test 6: Finetuning One Convolutional Layer and Weights Regularization

In this paragraph we exploited the conclusion mentioned in test 4, introducing here *L1_L2 weight regularization* on the one convolutional layer finetuned network.

Finetuning one conv layer and weights regularization				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
22	0.7487	0.7471	21.5590	9.8097



(a) Test 6 Accuracy

(b) Test 6 Loss

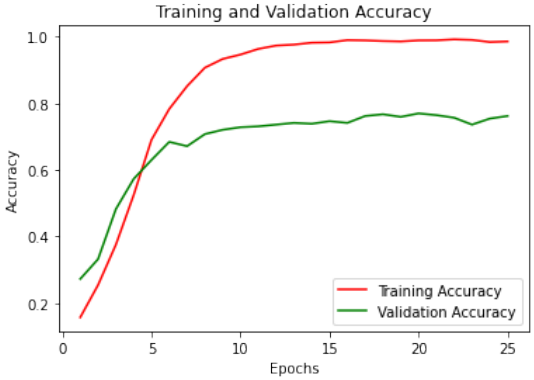
The shapes of the graphs are more or less the same of the two had in test 4, the only things that changes are the values of the losses which are here higher and more curvilinear due to the weight regularization approach used. Anyway, this heavy regularization helped us to surpass the result in test 1, not by much but it is an improvement that, maybe, can be exploited

finetuning more. Following this lead, in the next paragraph we use weight regularization with two convolutional layers finetuned.

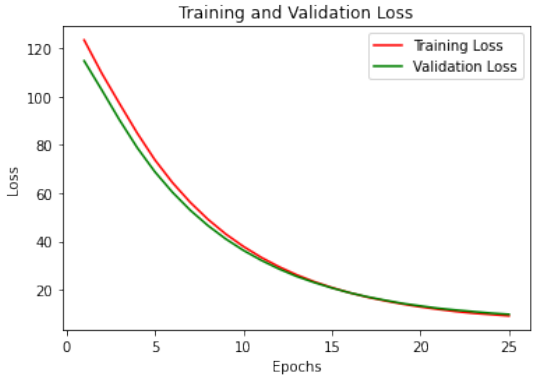
4.1.7 Test 7: Finetuning Two Convolutional Layers and Weights Regularization

Following the good result obtained in test 6, in this paragraph we add, to the network used in test 5, *L1_L2 weight regularization*.

Finetuning two conv layers and weights regularization				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
25	0.7694	0.7494	15.6742	9.8097



(a) Test 7 Accuracy



(b) Test 7 Loss

Even if the increase the flexibility of our network finetuning more layers, as hoped, we achieved a better accuracy which is also the best obtained so far.

4.1.8 Test 8: Genetic Algorithm for Hyper-parameters and Architecture Optimization

Following the elements presented in chapter 2.6 regarding the components of genetic algorithms, we here introduce the specific cases used here.

4.1.8.1 Genotype

In our specific case, our genes are bounded real-valued encoded and they represent three different parameters:

- *activation_function*: bounded between 0 and 1.999. The integer part stands for ReLU (0) and ELU (1);
- *optimizer*: bounded between 0 and 1.999. The integer part stands for rmsprop (0) and adam (1);
- *learning rate*: bounded between 0.001 and 0.1.

4.1.8.2 Population

Since for each individual we train a model and evaluate its performance, we decided to use only 5 individuals per generation to limit the computation.

4.1.8.3 Fitness Function

In our problem, the fitness function is the the function which calculate the maximum validation accuracy reached by the model. Our objective is maximizing the validation accuracy.

4.1.8.4 Selection Algorithm

The selection algorithm used in this case is *tournament selection*. In each round of the tournament selection method, two individuals are randomly picked from the population, and the one with the highest fitness score wins and gets selected. We decided to select only two individuals since our population is small and selecting more could cause an abuse in exploitation.

4.1.8.5 Crossover Algorithm

There are multiple algorithms applicable to real-value encoded individuals, we decided to use the *Simulated Binary Bounded Crossover*, which is a bounded version of the Simulated Binary Crossover (SBX)[13].

The idea behind the simulated binary crossover is to imitate the properties of the single-point crossover that is commonly used with binary-coded chromosomes. One of these properties is that the average of the parents' values is equal to that of the offsprings' values. When applying SBX, the two offspring are created from the two parents using the following formula:

$offspring_1 = \frac{1}{2}[(1+\beta)parent_1 + (1-\beta)parent_2]$

$offspring_2 = \frac{1}{2}[(1-\beta)parent_1 + (1+\beta)parent_2]$

Here, β is a random number referred to as the *spread factor*.
This formula has the following notable properties:

- The average of the two offspring is equal to that of the parents, regardless of the value of β.
- When the β value is 1, the offspring are duplicates of the parents.
- When the β value is smaller than 1, the offspring are closer to each other than the parents were.
- When the β value is larger than 1, the offspring are farther apart from each other than the parents were.

The probability to mate is set equal to 0.9[14].

4.1.8.6 Mutation Algorithm

As mutation algorithm we decided to use the *Polynomial Bounded* method, which is a bounded mutation operator that uses a polynomial function for the probability distribution. The probability to mutate is set equal to 0.5.

4.1.8.7 Elitism

We set the number of individuals for the elitism mechanism to 1 since our population is only made by 5 individuals. Anyway, to speed up even more the computation of the algorithm we introduced a system that saves the results of individuals already seen, in this way we do not need to recompute the accuracy again for those individuals.

[13] https://content.wolfram.com/uploads/sites/13/2018/02/09-2-2.pdf

[14] Value suggested by the book "Hands on Genetic Algorithms with Python" by Eyal Wirsansky, also the other values of probabilities are taken by the suggestions of the book

4.1.8.8 Results

Even if the code was uploaded on the directory, we was unable to run it till completion due to the limitations of Colab. In fact, even the strong limitation we made on the population, for each epoch the algorithm takes about 40 minutes (since all the first epochs are done at the same time, due to the parallelism introduced by *deap*), thus producing the final result (i.e., the result at the end of the last generation) after few tens of hours[15].

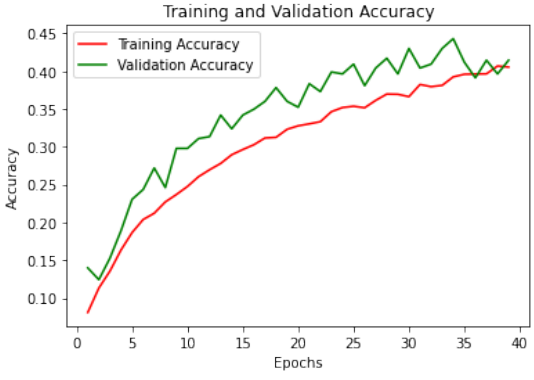
4.2 ResNet50V2

ResNet stands for Residual Network. It is an innovative neural network that was first introduced by Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun in their 2015 computer vision research paper titled "Deep Residual Learning for Image Recognition". This model was immensely successful, as can be ascertained from the fact that its ensemble won the top position at the ILSVRC 2015 classification competition with an error of only 3.57%. Additionally, it also came first in the ImageNet detection, ImageNet localization, COCO detection, and COCO segmentation in the ILSVRC & COCO competitions of 2015.

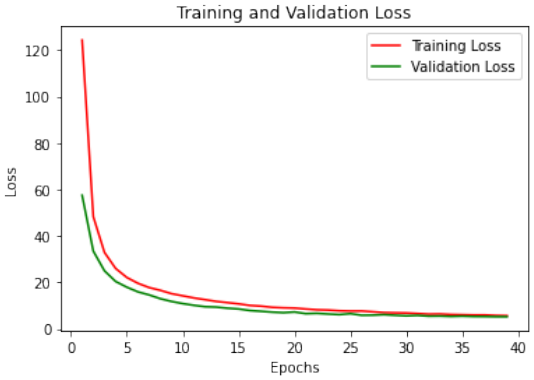
4.2.1 Test 1: Classical ResNet50V2 (Feature Extraction)

The original ResNet50 comes with a GlobalAveragePooling2D and a prediction layer soon after. In this test we used the same approach, resizing the prediction layer to the number of classes we have.
The results obtained after training are:

Feature Extraction				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
39	0.4430	0.3241	5.3733	6.8513



(a) Simple ResNet50V2 Feature Extraction Accuracy



(b) Simple ResNet50V2 Feature Extraction Loss

The network poorly performed compered with the results obtained using VGG16. This can be explained since the *imagenet dataset* is not specialized in distinguish paintings' artists, thus the frozen weights of our networks are very far to be the ones we need. To overcome this problem we can fine tune or add more layers after the *GlobalAveragePooling2D* layer.

Another problem can be notice looking at the accuracy plot, the training accuracy is less than the validation accuracy. To understand this issue, it is important to consider the difference of

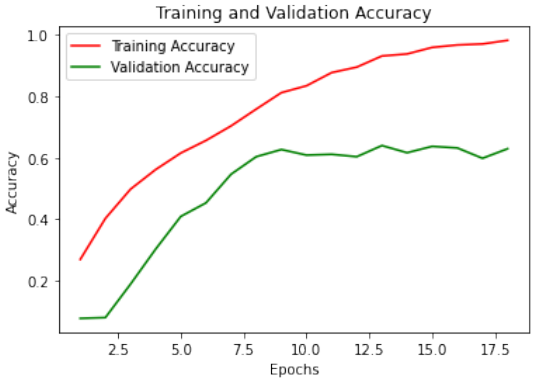
[15]Consider 40 minutes per epoch and an average of 18 epochs, we get that we need 720 minutes for training a generation, since we have 5 generations, the computation end after approximately 3600 minutes, which are 60 hours.

the number of images in the training and validation set. The validation set is made of about few hundreds of pictures against the few thousands of the training set, the latter may involve, with greater probability, paintings being part of a different period of the artists (e.g., the example shown in Image1), which can lead to a worst accuracy.

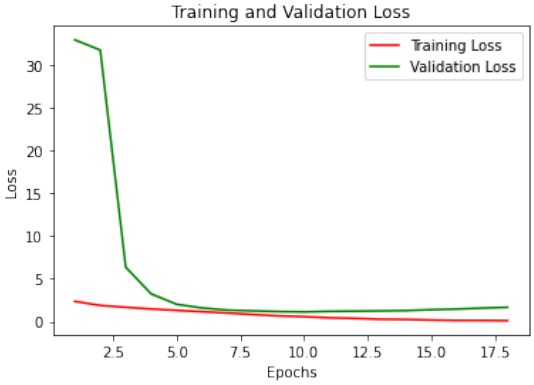
4.2.2 Test 2: Finetuning 1 block

ResNet50V2 is made of multiple big blocks (i.e., 5) so called *conv* in the model. These blocks are made of sub-blocks connected by each other by an add layer which connects the processed input (e.g., processed by Conv2D, Padding, Pooling, BatchNormalization) and the residual input[16]. We finetuned considering this under-blocks, hence in this paragraph we finetuned the *conv5_block3*.

Finetuning 1 block				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
18	0.6399	0.5793	1.2453	1.7916



(a) ResNet50V2 Test 2 Accuracy



(b) ResNet50V2 Test 2 Loss

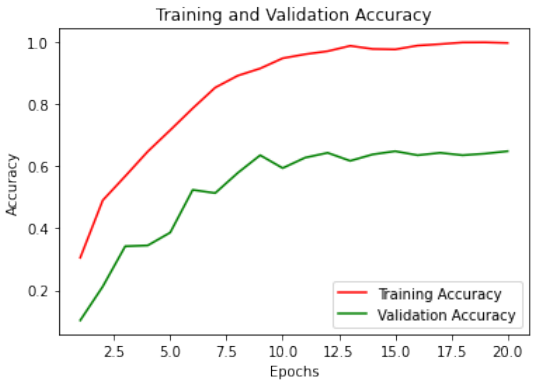
As expected, the network has definitely reach a more satisfactory result. Also the behaviors of the curves now is more reasonable.

4.2.3 Test 3: Finetuning 2 blocks

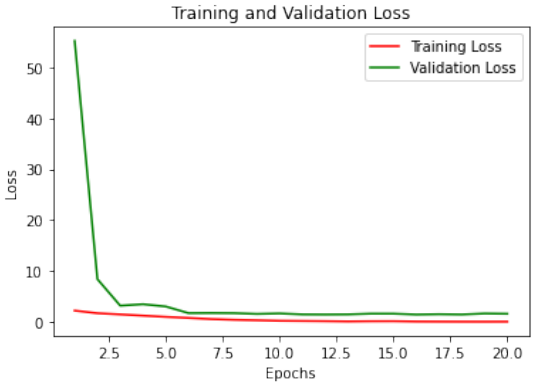
Since finetuning a block led to better results, the next step we did was to tuning also the previous sub-block, thus starting to tuning from *conv5_block2*.

Finetuning 2 blocks				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
20	0.6477	0.6092	1.5918	1.7734

[16]Sometimes the residual is downsampled using a max pooling layer, this is done in order to match the actual size of the feature map obtained at that level of the network.



(a) ResNet50V2 Test 3 Accuracy



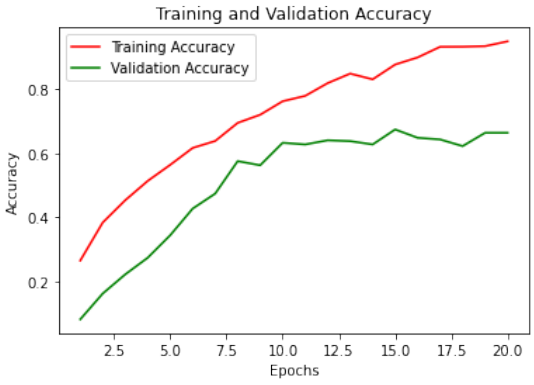
(b) ResNet50V2 Test 3 Loss

This results proves that our idea of finetuning was good since the network improved a little. Anyway, finetuning more can be risky since the computational power and time needed can be very high and becoming unbearable to the limits imposed by Colab. Thus, we tried to improved (without satisfaction, as the nexts to paragraphs describe) our network adding dense layers between our *GlobalAveragePooling2D* and our prediction layer, following the structure used in the VGG16 study.

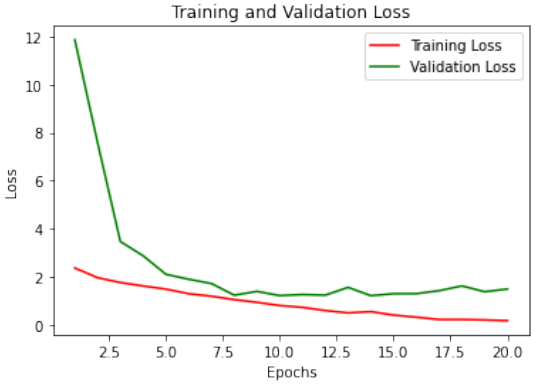
4.2.4 Test 4: Finetuning with One Block and Adding Two Dense layers

Here we took the same approach of test 3, but adding two dense layers followed by a dropout layer (as used in figure7).

Finetuning one block and dense layers				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
20	0.6736	0.6368	1.6655	1.7734



(a) ResNet50V2 Test 4 Accuracy



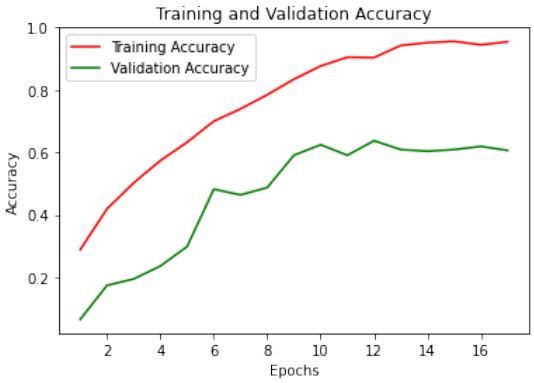
(b) ResNet50V2 Test 4 Loss

Enlarging in this way the network resulted in better performance. This can be because the dense layers before the softmax can better exploit the information of the *GlobalAverage-Pooling2D*, keeping the problem for more time to a higher dimensionally than the one used for prediction.

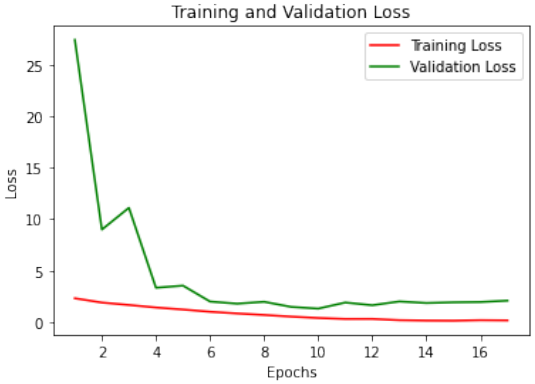
4.2.5 Test 5: Finetuning with Two Blocks and Adding Two Dense layers

In this paragraph we did the same thing done in test 4, but finetuning two blocks as in test 3.

Finetuning two blocks and dense layers				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
17	0.6373	0.5954	1.6404	2.1089



(a) ResNet50V2 Test 5 Accuracy



(b) ResNet50V2 Test 5 Loss

Unfortunately, finetuning this much with these additional layers led to worst performance. Perhaps, this is due to backpropagation, which changed the weights (which are 9M, against the 14M that are not trainable) to a not-optimal solution. Anyway, this result is also too poor compared to test 3, which without having the newly added dense layers still perform better.

4.3 ResNet101V2

ResNet101 was implemented as one of the model used in the ensemble method cited in the ResNet50V2 introduction. ResNet101 is an extension of ResNet50 that goes deeper reaching 101 layers in depth, simply adding more convolutional 'sub'-blocks in the blocks[17]. ResNet101 proved to be slightly more accurate than ResNet50.[18]

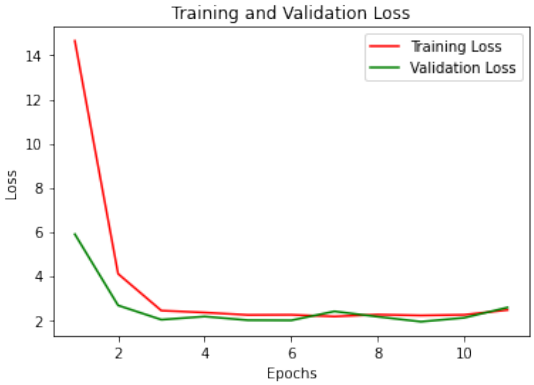
4.3.1 Test 1: Classical ResNet101V2

The original ResNet50 comes with a GlobalAveragePooling2D and a prediction layer soon after. In this test we used the same approach, resizing the prediction layer to the number of classes we have.
The results obtained after training are:

Feature Extraction				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
11	0.3756	0.3195	1.9798	2.6044

[17] Referring to the notation used before, we intend as a sub-block a block between two adds.
[18] It is not only additional network used in the ensemble method, an extension of it was also used, ResNet152. This model is the more accurate of the three, but because of it size we decide to limit our studies to the 101-layers version.

(a) ResNet101V2 Feature Extraction Accuracy



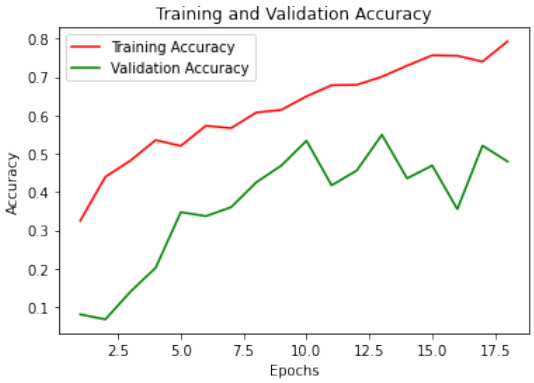
(b) ResNet101V2 T Feature Extraction Loss

The network under-fitted our training, performing drastically poorly. The same considerations done in chapter 4.2.1 apply here.

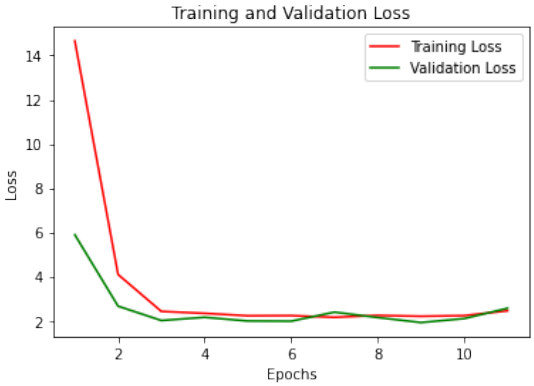
4.3.2 Test 2: Finetuning One Sub-Block

As done for ResNet50, we tried to improve our ResNet101 performance using finetuning. In this paragraph we just finetuned the last sub-block of the network.

Finetuning One Sub-Block				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
18	0.5492	0.5218	1.7628	2.3036



(a) ResNet101V2 Test 2 Accuracy



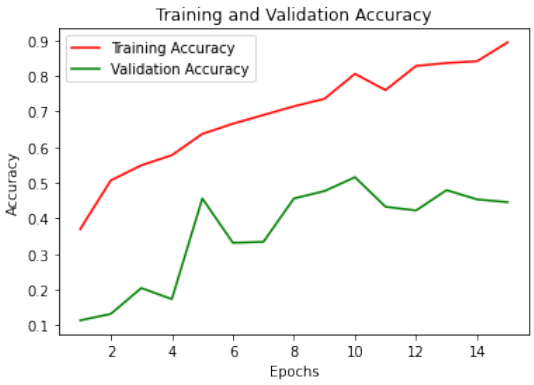
(b) ResNet101V2 Test 2 Loss

As expected, the performance are better compered to test 1. Anyway, the network still do not overfit: finetuning just one sub-block is not enough for a neural network made of 101 layers.

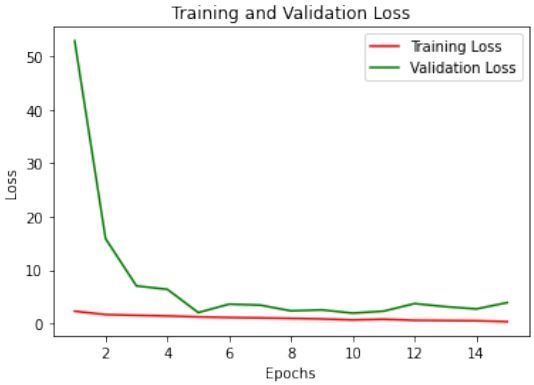
4.3.3 Test 3: Finetuning the Entire Block 5

Aiming to overfitting, we finetuned the entire block 5, which is made of 3 sub-blocks.

Finetuning Entire Block 5				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
15	0.5155	0.4805	1.9216	3.4421



(a) ResNet101V2 Test 3 Accuracy



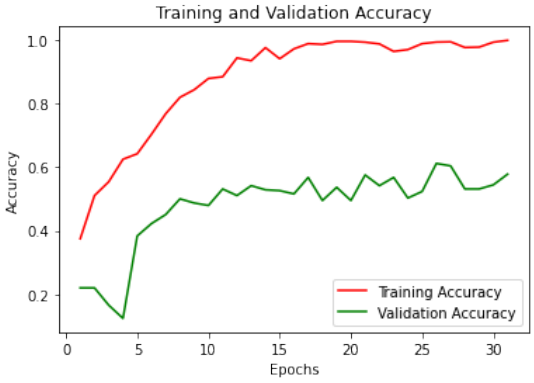
(b) ResNet101V2 Test 3 Loss

Unfortunately, we have little less performance and we still not completely overfitting: the training accuracy is less than 0.9, meaning that we are not finetuning enough but we are following the right lead since the training accuracy gain 0.1 point.

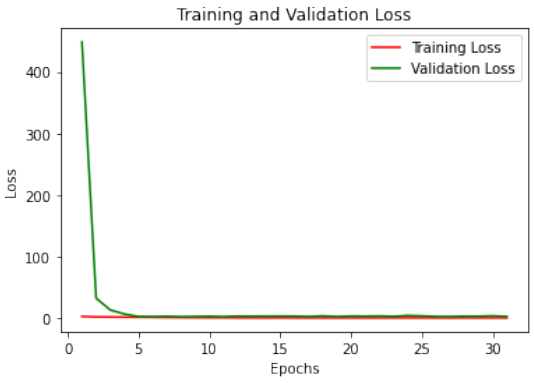
4.3.4 Test 4: Finetuning Half Block 4

In this paragraph, heavy finetuning is done setting as trainable parameters all of them after *conv4_block13_out* for a total of 13 sub-blocks finetuned (i.e., 10 in block 4 and 3 in block 5).

Finetuning Half Block 4				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
31	0.6114	0.5977	2.3865	2.3642



(a) ResNet101V2 Test 4 Accuracy



(b) ResNet101V2 Test 4 Loss

Finetuning these many layers helped the network to be more our topic-specific. In fact, the neural network finally overfit properly our training data as the accuracy plot shows, also even the loss is lesser than before. However, even if we improve from the previous test it is not enough if we consider the best result obtained using ResNet50 or VGG16.

4.3.5 Test 5: Test 4 and Dropout

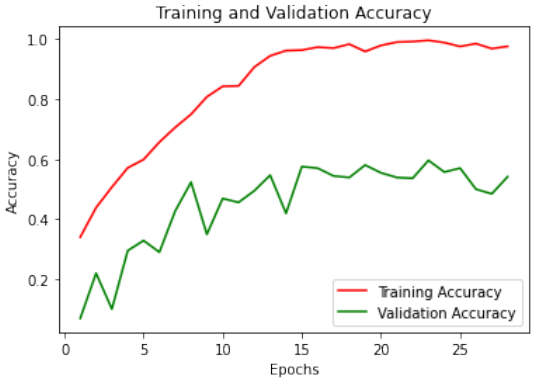
In test 4 the network overfitted, even if not much, we try to use dropout to mitigate this problem. This approach can result in two possible outcomes:

1. We perform more or less the same as in test 4: this is the goal of dropout, neutralize neurons but performing the same as before, reducing in this way the number of computations and performing better on the test set.

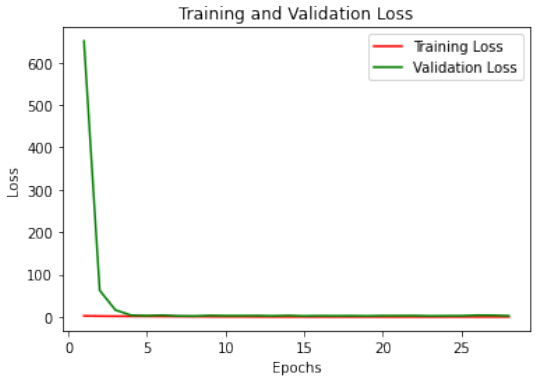
2. We perform worst than test 4: dropout neutralizes too many units, performing worst both on the validation set and the test set. This is a problem, the dropout rate should be reduced or other regularization techniques apply.

The results obtained are the following:

Finetuning Half Block 4 and Dropout				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
28	0.5959	0.5356	2.2455	2.8590



(a) ResNet101V2 Test 5 Accuracy



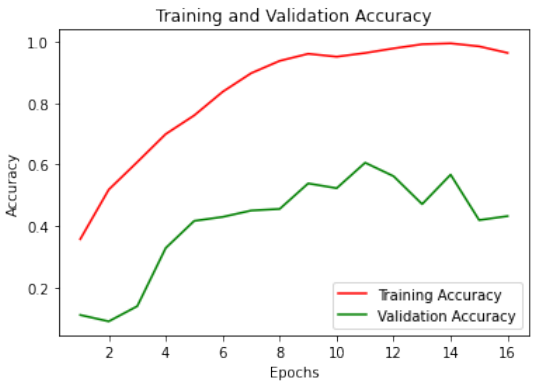
(b) ResNet101V2 Test 5 Loss

The plots and the table show that the situation this test inclines is the second outcome presented. Even if the validation accuracy losses just 0.2 points, the test accuracy has a gap 0.6 points with the one obtained in test 4, which is a lot considering that the overall performance is not good (it barely arrives to 0.6, which is very low to be considered a good classifier). Hence, for the following test we decided to drop the use of dropout until better performances are met.

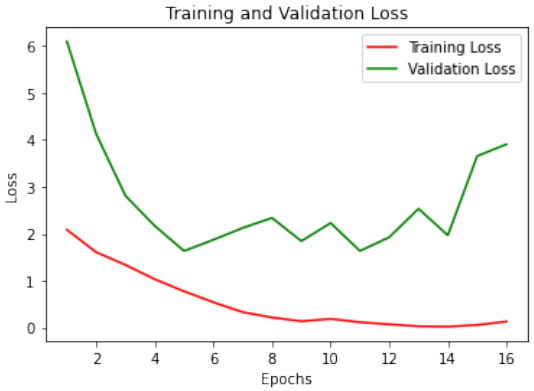
4.3.6 Test 6: Adding Dense Layers to Test 4

Following the tests performed with VGG16 and ResNet50V2, we tried also here to add dense layers at the top of the *GlobalAveragePooling2D* layer. Doing this experiment, the learning rate of *Adam* is change from the default value (i.e, 0.001) to 0.0001.
The result obtained are the following:

Finetuning Half Block 4 and Adding 2 Dense Layers				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
16	0.6062	0.4552	1.6339	3.5964



(a) ResNet101V2 Test 6 Accuracy



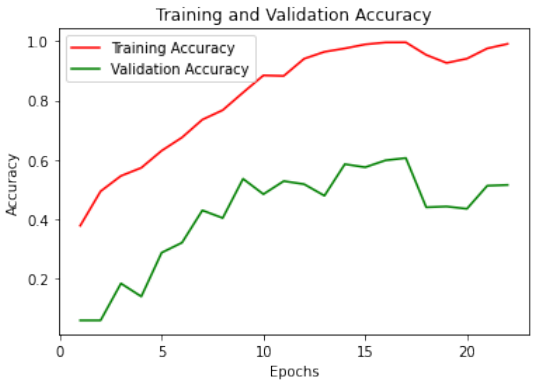
(b) ResNet101V2 Test 6 Loss

The introduction of these layers worsen the performance of test 4, but they helped achieving better loss values. Anyway, they had a bad effect on the test accuracy which is now one of the lower values obtained for ResNet101. This can be explained, perhaps, since the network now shrinks in two layers before predicting: layers specialized on the training set. Hence, they ended up discovering features that are not helpful for predicting correctly the test set.

4.3.7 Test 7: Going Deeper and Deeper

Since increasing the finetuning level led to better result in the previous tests, we finetuned ResNet101 till *conv3_block4_out* including the parameters of all conv4 (i.e., block4). The network in this way has really unbalanced in terms of trainable parameters, most of them were trainable (i.e., about 80%). Even that, we tested it and what we obtain was a failure, the network didn't learn well an ended up with a tremendous low accuracy. Thus, we decided to limit our going deeper to the *conv4_block10_out*: adding three more sub-blocks to the ones tested in test 3 (which is still our current optimum). The result obtained are the following:

Finetuning till conv4_block10_ou				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
22	0.6062	0.5310	2.1705	2.5997



(a) ResNet101V2 Test 7 Accuracy

(b) ResNet101V2 Test 7 Loss

These results gained the second best position for our ResNet101 tests. Anyway, the performance are still too poor and the computational power required for training is very high (i.e., we have about 30M parameters). It is important to notice that the same result for validation was obtained in test 6, but with a lower score for the test accuracy. This indicates that the point is a pitfall, where our validation gets caught and wherever it moves it finds only worst values.

Anyway, we have no way to escape it: the accuracy for the training set is curvilinear without frequent zig-zags, thus changing the learning rate of Adam will let to worst or the same result:

1. if we decrease the learning rate we increase the possibility to stop earlier or to fall in the same pitfall;

2. if we increase the learning rate we will overfit very fast jeopardizing the training.

4.4 InceptionV3

InceptionV3 is a widely-used 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 based on the original paper: "Rethinking the Inception Architecture for Computer Vision" by Szegedy, et. al.

The model itself is made up of symmetric and asymmetric building blocks, including convolutions, average pooling, max pooling, concats, dropouts, and fully connected layers. Batchnorm is used extensively throughout the model and applied to activation inputs. Loss is computed via softmax.

A high-level diagram of the first model proposed is shown below:

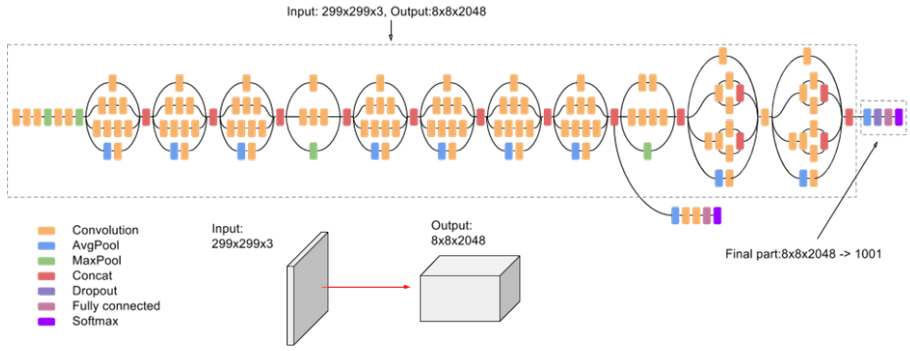


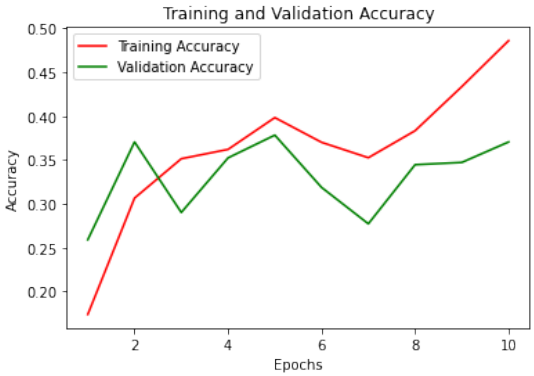
Figure 26: Inception v3 First Proposal

Anyway, the model presented in figure is different to the one present in Keras, which at the end of the network has simply a *GlobalAveragePooling* layer and a prediction layer. The model depicted has a more complex output and it make use of an *auxiliary classifier*: training using loss at the end of the network didn't work well since the network is too deep and the gradients don't propagate cleanly. As a hack, the idea of the time was to attach "auxiliary classifiers" at several intermediate points in the network that also try to classify the image and receive loss. All of that just because the network was proposed before *batch normalization*, with batch normalization there is no longer need to use this trick, In fact, the current version of InceptionV3 implemented in Keras uses BatchNorm.

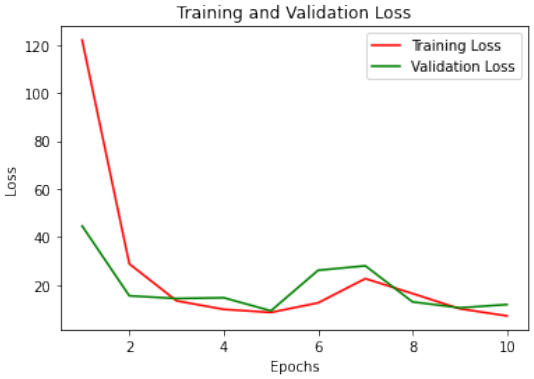
4.4.1 Test 1: Classical InceptionV3

The base InceptionV3 provided by Keras comes with a GlobalAveragePooling2D and a prediction layer after that. In this test we used the same approach, resizing the prediction layer to the number of classes we have.
The results obtained after training are (learning rate=0.01, Adam):

Feature Extraction				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
10	0.3782	0.4092	9.3051	10.9338



(a) InceptionV3 Test 1 Accuracy



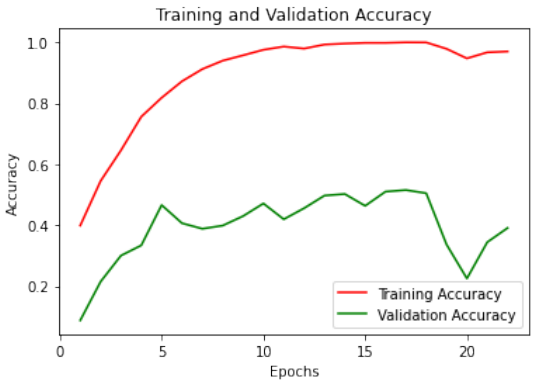
(b) InceptionV3 Test 1 Loss

The network under-fitted our training, performing drastically poorly. The same considerations done in chapter 4.2.1 apply here.

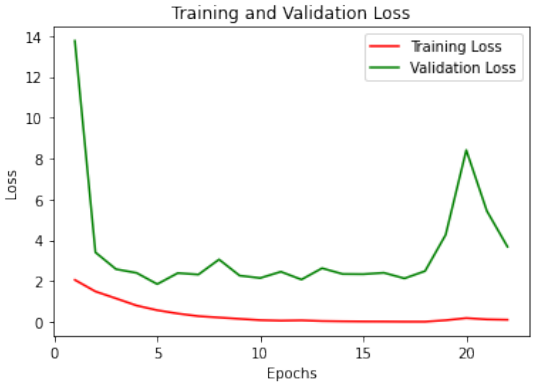
4.4.2 Test 2: Finetuning 1 Block

As done for the previous networks, we finetuned one block at a time, where a block is an inception block.
The results for finetuning only the last block are (learning rate=0.001, Adam):

Finetuning 1 Block				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
22	0.5155	0.4437	2.1245	3.3541



(a) Inception Test 2 Accuracy



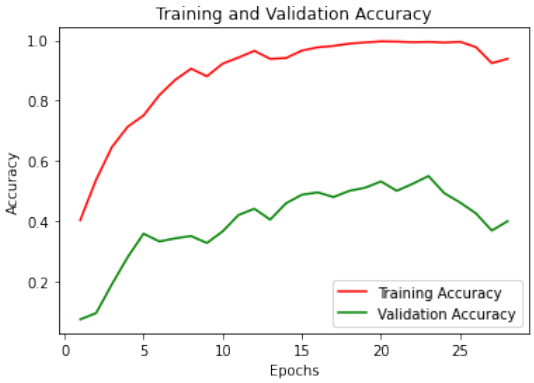
(b) Inception Test 2 Loss

As expected, the network finally overfitted and the accuracy of the validation set increased as well. Anyway, the curves in the last epochs have a strange behavior, the validation accuracy and loss suddenly go worst. Perhaps, this is due to the fact that we are overfitting by few epochs at that point and the iteration moves the curves to points that are in the close to an high accuracy, but which features are far different from the one of the validation set.

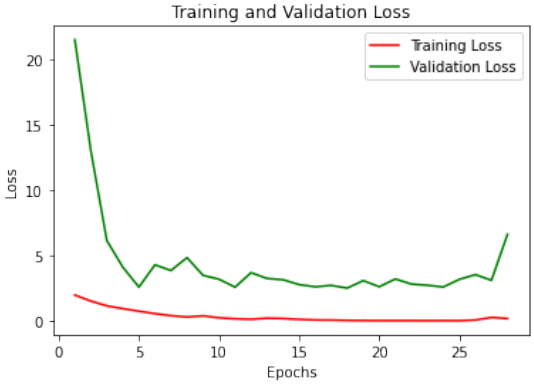
4.4.3 Test 3: Finetuning 2 Blocks

Since we want to find first a good number of blocks to set trainable and than optimize it, we continued our test finetuning the last 2 blocks.
The results obtained are (learning rate=0.001, Adam):

Finetuning 2 Blocks				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
28	0.5492	0.4184	2.7196	6.3845



(a) Inception Test 3 Accuracy



(b) Inception Test 3 Loss

The plots are more or less similar to the ones in the previous test, but the validation accuracy is greater by 0.3 points and the test accuracy is less than 0.3 points. As said before, this can be addressed to the fact that the test and validation set can be different to each other in term of style, thus same feature discovered by a feature map can apply correctly to one case and not in the other. Anyway, since we are supposed to know nothing about the test set and our conclusion should only be driven by the result of the validation, the model can be consider an improvement of the one obtained in the previous test.

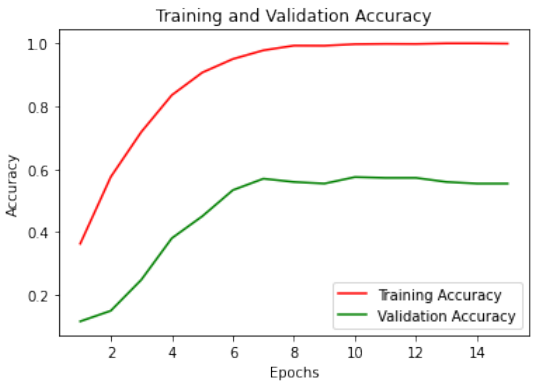
Also in this case, the accuracy (loss) decreases (increases) suddenly, but the behavior seems here to be alleviated. Thus, suggesting that finetuning more blocks can help preventing this strange behavior.

4.4.4 Test 4: Finetuning 3 Blocks

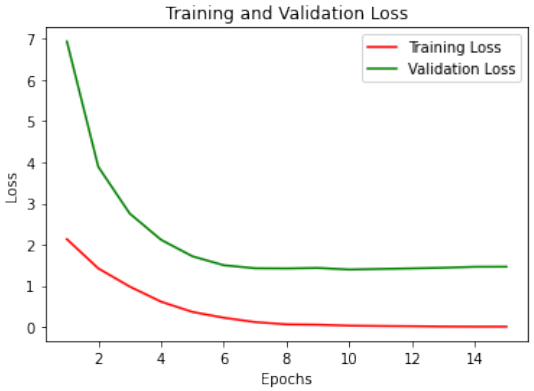
Finetuning till the third last block was our maximum in finetuning this type of network, since finetuning more led to worst performance[19].
The results obtained are (learning rate=0.0001, Adam):

Finetuning 3 Blocks				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
15	0.5725	0.5908	1.4111	1.4673

[19]Here not reported, but the code on GitHub provide the possibility to finetune other 2 more layers.



(a) Inception Test 4 Accuracy



(b) Inception Test 4 Loss

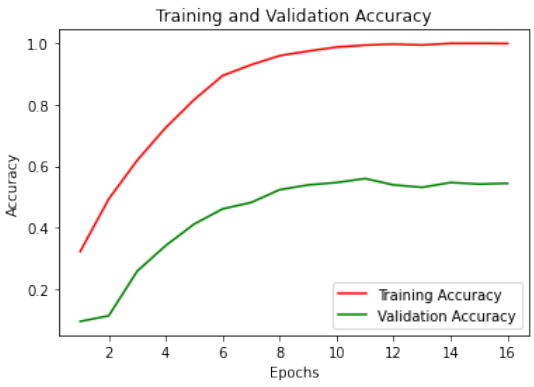
First thing to notice is the absence of the strange behavior underlined in the previous paragraphs, having more parameters to train help our network to discover a better solution. The validation accuracy gained other 0.3 points and the testing accuracy increases a lot reaching 0.59: the model become more precise in understanding the differences between authors, differences that in the testing loss are more concise reading this result.

Anyway, we overfit very fast so we decided to try preventing it using dropout.

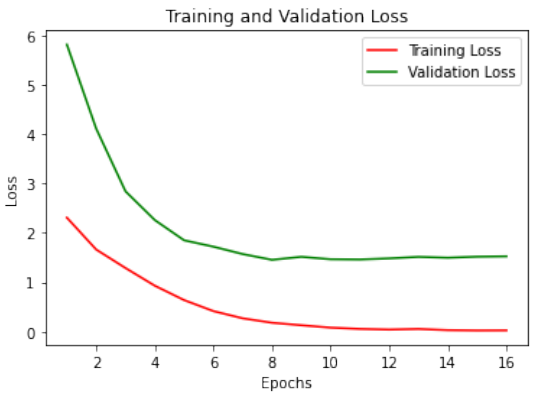
4.4.5 Test 5: Finetuning 3 Blocks with Dropout

In this test, we added a dropout layer after the *GlobalAveragePooling2D*
The results obtained are (learning rate=0.0001, Adam):

Finetuning 3 Blocks with Dropout				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
16	0.5596	0.6069	1.4603	1.3284



(a) Inception Test 5 Accuracy



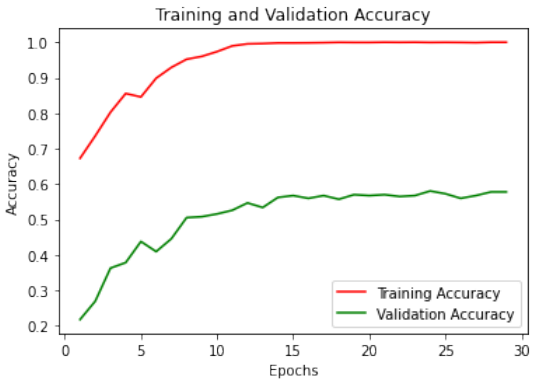
(b) Inception Test 5 Loss

The plots are quite the same of the test without dropout, but we need an additional epoch to stop the training. The dropout effect is very low, this is quite reasonable since the most of units are inside the InceptionV3 architecture. Anyway, by pure luck we achieve our best test accuracy till now, but since, at this point of the study, we care only about the validation accuracy, this result is meaningless.

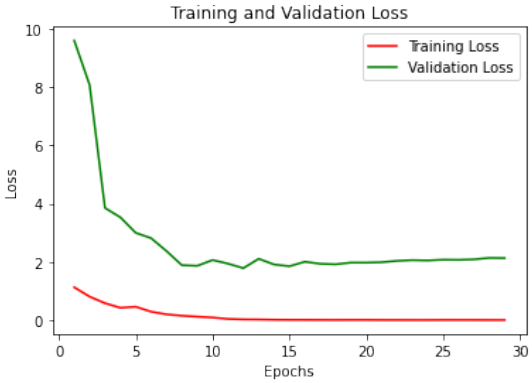
4.4.6 Test 6: Finetuning 3 Blocks and ExponentialDecay Learning Rate

In this test, we perform the same strategy of test 4 (finetuning 3 blocks), but we changed the optimizer learning rate. In order to try different approaches, here we used Adam with a *exponential learning rate decay*, which can be used through the **ExponentialDecay** class provided by Keras. The constructor takes 3 parameters[20]: the initial learning rate, the decay step and the decay rate, which we set to 0.01, 25 and 0.9 respectively.

Finetuning 3 Blocks and ExponentialDecay Learning				
Epoch stopped	**Validation Accuracy**	**Testing Accuracy**	**Validation Loss**	**Testing Loss**
29	0.5803	0.6253	2.0511	2.3217



(a) Inception Test 6 Accuracy



(b) Inception Test 6 Loss

It is important to notice that this method led us to reach high accuracy values on the training very quickly, but at the same time increasing the validation accuracy rapidly, and then stabilizing and growing slightly (almost imperceptibly looking at the plot), leading to improve more and more the accuracy obtaining the reported value. Moreover, we have a very high value for the test set as well, making this model the best obtained for InceptionV3.

An important thing to take into consideration is the behavior of the loss, which appears here much higher than in the fourth test, this can be explained perhaps by a different direction that our system takes in the search for convergence.

[20]For the others we used the default values

5 — Ensemble Network

Ensembling consists of pooling together the predictions of a set of different models to produce better predictions. Nowadays, in every machine learning competitions, in particular on Kaggle, the winners use very large ensembles of models that inevitably beat any single model, no matter how good, hence we decided to use it also in our project.

Ensembling relies on the assumption that different well-performing models trained independently are likely to be good for different reasons: each model looks at slightly different aspects of the data to make its predictions, getting part of the "truth" but not all of it[21]. By pooling the perspective of different models together, a far more accurate description of the data can be obtained.

5.1 Our approach

In our project we decided to use a simple ensembling approach, which is *average voting*: average voting computes the average of the scores (i.e., softmax output) of the base classifiers and select the class with the highest score.

Although we decided to use this simple approach, even if it is very common, we decided to optimize the choice of models to be used in order to maximize the accuracy obtained on the testset. In fact, all the models described in the chapter "pre-trained models" have been saved as *".keras"* models so that they can be reloaded and used to directly evaluate the testset in the ensemble, so that they do not need to be trained again (thus increasing the speed of test execution).

The main problem was to choose which of these models combined together obtained the best accuracy. Since the number of models was more than 20 a grid search seemed too slow as a method of solving this problem, so we decided to use a genetic algorithm to solve the problem automatically.

Anyway, due to the fact that we have different inputs for each type of network (ResNet, VGG and Inception) we decided to divide the ensembling networks into three types, one for each type of network.

5.1.1 The Ensemble Classese

To perform the ensemble in an orderly fashion, what was done was to create three classes called respectively *EnsembleVGG*, *EnsembleResNet* and *EnsembleInception* . The classes allows to construct and ensemble method based on a subset or the entire set of network provided to it, and evaluate the obtained network using the *Keras* function *evaluate*.

The class for VGG16 models is as follows:

```
class EnsembleVGG:
    def __init__(self, test_images_vgg):
        self.test_images_vgg = test_images_vgg

        # create list of models
        self.model_list = []

        base_dir = './models/'
        # VGG16
        for name in vgg16:
            tmp = ks.models.load_model(base_dir + 'vgg16/' + name)
            tmp._name = name
            for i, layer in enumerate(tmp.layers):
                layer.trainable = False
```

[21] This can be see in detail in the "data visualization" chapter

```
                layer._name = 'ensemble_' + str(i + 1) + '_' + layer.name
            self.model_list.append(tmp)


    def ensembleModel(self, active_models):
        models = []
        for i, model in enumerate(self.model_list):
            if active_models[i] == 1:
                model._name = str(i)
                models.append(model)
        model_input = ks.Input(shape=(IMAGE_WIDTH_VGG, IMAGE_HEIGHT_VGG, 3))
        model_outputs = [model(model_input) for model in models]
        ensemble_output = ks.layers.Average()(model_outputs)
        ensemble_model = ks.Model(inputs=model_input, outputs=ensemble_output)
        ensemble_model.compile(metrics=['accuracy'])
        return ensemble_model

    def evaluate(self, ensemble_model):
        loss, acc = ensemble_model.evaluate(self.test_images_vgg)
        return acc

    def ensemble_and_evaluate(self, active_models):
        if np.sum(active_models) == 0:
            return 0.0
        elif np.sum(active_models) == 1:
            print(active_models.index(1))
            return self.evaluate(self.model_list[active_models.index(1)])
        else:
            return self.evaluate(self.ensembleModel(active_models))

    def models_len(self):
        return len(self.model_list)
```

The class related to the ResNet family of networks is as follows:

```
class EnsembleResNet:
    def __init__(self, test_images_resnet):
        self.test_images_resnet = test_images_resnet

        # create list of models
        self.model_list = []

        base_dir = '/content/drive/MyDrive/Fazzari_Ramo/Models/'
        # ResNet
        resnet = resnet50 + resnet101
        for i, name in enumerate(resnet):
            if i < len(resnet50):
                tmp = ks.models.load_model(base_dir + 'resnet50/' + name)
            else:
                tmp = ks.models.load_model(base_dir + 'resnet101/' + name)
            tmp._name = name
            for j, layer in enumerate(tmp.layers):
                layer.trainable = False
                layer._name = 'ensemble_' + str(j + 1) + '_' + layer.name

            self.model_list.append(tmp)



    def ensembleModel(self, active_models):
        models = []
        for i, model in enumerate(self.model_list):
            if active_models[i] == 1:
                model._name = str(i)
```

```
                models.append(model)
        model_input = ks.Input(shape=(IMAGE_WIDTH_RESNET, IMAGE_HEIGHT_RESNET,
                                      3))
        model_outputs = [model(model_input) for model in models]
        ensemble_output = ks.layers.Average()(model_outputs)
        ensemble_model = ks.Model(inputs=model_input, outputs=ensemble_output)
        ensemble_model.compile(metrics=['accuracy'])
        return ensemble_model

    def evaluate(self, ensemble_model):
        loss, acc = ensemble_model.evaluate(self.test_images_resnet)
        return acc

    def ensemble_and_evaluate(self, active_models):
        if np.sum(active_models) == 0:
            return 0.0
        elif np.sum(active_models) == 1:
            print(active_models.index(1))
            return self.evaluate(self.model_list[active_models.index(1)])
        else:
            return self.evaluate(self.ensembleModel(active_models))

    def models_len(self):
        return len(self.model_list)
```

Finally, the ensemble class for Inception is:

```
class EnsembleInception:
    def __init__(self, test_images_inception):
        self.test_images_inception = test_images_inception

        # create list of models
        self.model_list = []

        base_dir = '/content/drive/MyDrive/Fazzari_Ramo/Models/'
        # VGG16
        for name in inception:
            tmp = ks.models.load_model(base_dir + 'inception/' + name)
            tmp._name = name
            for i, layer in enumerate(tmp.layers):
                layer.trainable = False
                layer._name = 'ensemble_' + str(i + 1) + '_' + layer.name
            self.model_list.append(ks.models.load_model(base_dir + 'inception/'
                                   + name))


    def ensembleModel(self, active_models):
        models = []
        for i, model in enumerate(self.model_list):
            if active_models[i] == 1:
                model._name = str(i)
                models.append(model)
        model_input = ks.Input(shape=(IMAGE_WIDTH_INCEPTION,
                                      IMAGE_HEIGHT_INCEPTION, 3))
        model_outputs = [model(model_input) for model in models]
        ensemble_output = ks.layers.Average()(model_outputs)
        ensemble_model = ks.Model(inputs=model_input, outputs=ensemble_output)
        ensemble_model.compile(metrics=['accuracy'])
        return ensemble_model

    def evaluate(self, ensemble_model):
        loss, acc = ensemble_model.evaluate(self.test_images_inception)
        return acc

```

```
    def ensemble_and_evaluate(self, active_models):
        if np.sum(active_models) == 0:
            return 0.0
        elif np.sum(active_models) == 1:
            print(active_models.index(1))
            return self.evaluate(self.model_list[active_models.index(1)])
        else:
            return self.evaluate(self.ensembleModel(active_models))

    def models_len(self):
        return len(self.model_list)
```

The constructor takes as input the testset for which the ensemble class will do the evaluation, and what it does is load the different models. Next, in each three we have four functions:

1. *ensembleModel*: given an array of equal size to the number of different models, the function creates an ensemble model using the models that are in the position where in the input array there is a one.
2. *evaluate*: calls the keras evaluate function on the ensemble model
3. *ensemble_and_evaluate*: calls both the ensembleModel and the evaluate functions. In cases the input is all made of zeros, it returns 0 without farther computations, on the other hand if the input has only one 1 in it the function evaluates it directly.
4. *models_len*: retrieves the number of models.

Notice that in all functions we modified the name of the layers and models in order to prevent every form of name duplication in the ensemble network.

5.1.2 Genetic Algorithm Workflow

Following the guidelines in Section 2.6, we need to decide how to develop the different components and decide on the hyperparameters in order to define the flow of the genetic algorithm.

5.1.2.1 Genotype

In our specific case, our chromosomes are binary encoded and each gene represents a different model. Hence, we have individuals made of a number of genes equal to the number of different models that we have.

```
# create an operator that randomly returns 0 or 1
toolbox.register('zeroOrOne', random.randint, 0, 1)

# define a single objective, maximizing fitness strategy:
creator.create('FitnessMax', base.Fitness, weights=(1.0,))

# create the Individual class based on list:
creator.create('Individual', list, fitness=creator.FitnessMax)

# create the individual operator to fill up an Individual instance:
toolbox.register('individualCreator', tools.initRepeat, creator.Individual,
                                        toolbox.zeroOrOne, INDIVIDUAL_LENGTH)
```

5.1.2.2 Population

The population is composed of 8 individuals, we decided to keep the number of individuals low so as to reduce the number of evaluations done and, therefore, the total duration of the algorithm, in order to obtain a satisfactory result by waiting less than an hour.

```
toolbox.register('populationCreator', tools.initRepeat, list, toolbox.
                                        individualCreator)
```

5.1.2.3 Fitness Function

The fitness function is the accuracy obtained through the *predict_and_evaluate* function of the Ensemble class. It was registered in the *toolbox* as follows:

```
def ensembleAccuracy(individual):
    return ensemble.predict_and_evaluate(individual),

toolbox.register('evaluate', ensembleAccuracy)
```

Notice that the function returns a tuple, this is done because the framework accept only tuple values as result of the evaluation function.

5.1.2.4 Selection Algorithm

The selection algorithm used in this case is *tournament selection*. In each round of the tournament selection method, two individuals are randomly picked from the population, and the one with the highest fitness score wins and gets selected. We decided to select only two individuals since our population is small and selecting more could cause an abuse in exploitation.

```
# Tournament selection with tournament size of 2:
toolbox.register("select", tools.selTournament, tournsize=2)
```

5.1.2.5 Crossover Algorithm

As crossover algorithm we decided to use the *two-point crossover*. In the two-point crossover method, two crossover points on the chromosomes of both parents are selected randomly. The genes residing between these points are swapped between the two parent chromosomes.

The following diagram demonstrates a two-point crossover carried out on a pair of binary chromosomes, with the first crossover point located between the third and fourth genes, and the other between the seventh and eighth genes:

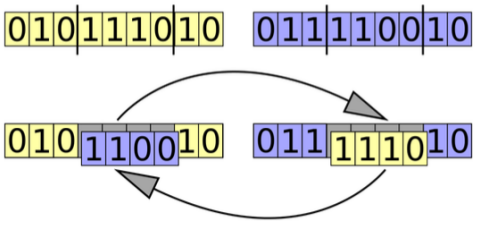


Figure 33: Two point crossover example.

The code for that is:

```
toolbox.register("mate", tools.cxTwoPoint)
```

5.1.2.6 Mutation Algorithm

As mutation algorithm we decided to use the *Multiple Flip bit mutation*. When applying the flip bit mutation to a binary chromosome, one gene is randomly selected and its value is flipped (complemented), as shown in the following diagram:

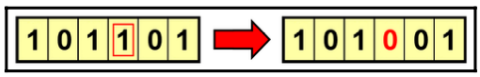


Figure 34: Flip bit mutation example.

This can be extended to several random genes being flipped instead of just one, obtaining the multiple flip bit mutation that we use. The code for that is:

```
toolbox.register("mutate", tools.mutFlipBit, indpb=1.0/INDIVIDUAL_LENGTH)
```

5.1.2.7 Elitism

We set the number of individuals for the elitism mechanism to 1. This value was decided to have a minimum of elitism, but without abusing it too much. In fact, the number of individuals is so low that having a higher value could lead to a problem, therefore, in order not to risk bad results (i.e., not global optimum results), we chose to use this conservative approach and limit the number to only one.

5.1.2.8 GA Flow

The first thing to do is to define the initial population, this is easily done by the following line of code:

```
population  = toolbox.populationCreator(n=POPULATION_SIZE)
```

After that, we created a statistical object. This object has served to have a report of the flow of the genetic algorithm, allowing us to save for each generation the maximum and average fitness values obtained, so that we can show them once the generations are over.

```
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("max", np.max)
stats.register("avg", np.mean)
```

Then, the last object we need to create for the *eaSimple* is the **HallOfFame**, which can be done through the following line of code:

```
hof = tools.HallOfFame(HALL_OF_FAME_SIZE)
```

The main flow is done using the *eaSimple* in the way as follow:

```
population, logbook = eaSimpleWithElitism(population,
                    toolbox,
                    cxpb=P_CROSSOVER,
                    mutpb=P_MUTATION,
                    ngen=MAX_GENERATIONS,
                    stats=stats,
                    halloffame=hof,
                    verbose=True)
```

Where the P_CROSSOVER is equal to 0.9, P_MUTATION 0.1.[22]

5.1.3 GA Results for VGG16 Ensemble

The following graph shows the results obtained for each generation from the population (i.e., the maximum and mean accuracy):

[22] Value suggested by the book "Hands on Genetic Algorithms with Python" by Eyal Wirsansky, also the other values of probabilities are taken by the suggestions of the book

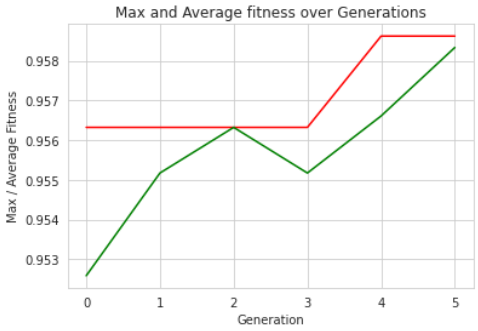


Figure 35: Maximum and average accuracy in the population for each generation.

As it can be noticed from the graph the elitism allows the maximum value to never go down. Moreover, it is possible to notice that the maximum result obtained is much higher than the one obtained with VGG16, which reached almost 80%, instead here we reach 95.86%. This was obtained from the genetic algorithm by combining the following models together:

- VGG16 feature extraction
- VGG16 feature extraction with dropout
- VGG16 with 2 layers finetuned and with dropout
- VGG16 with 1 layer finetuned

Chosen by the chromosome [0, 0, 1, 1, 1, 1, 0], our winner in the final population.

5.1.4 GA Results for ResNet Ensemble

The following graph shows the results obtained for each generation from the population (i.e., the maximum and mean accuracy):

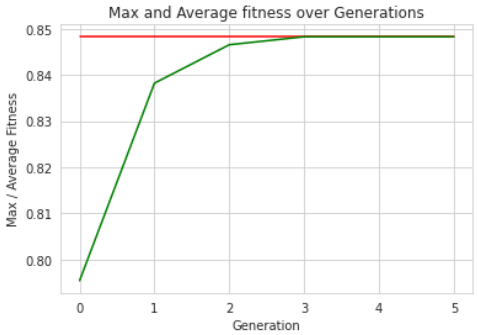


Figure 36: Maximum and average accuracy in the population for each generation.

The maximum accuracy value for ResNet was obtained in Section 4.2.4 using fine tuning of one block and two additional dense layers at the end of the network, reaching an accuracy score of 0.6736. Also here we exceed the value reached by the single model obtaining a much higher value, reaching 0.8483, satisfactory value even if lower than the one obtained with the ensemble of VGGs. This was obtained from the genetic algorithm by combining the following models together:

- ResNet50 with 1 block finetuned
- ResNet50 with 2 blocks finetuned

- ResNet50 with 1 block finetuned plus two dense layers at the end of it
- ResNet50 with 2 blocks finetuned plus two dense layers at the end of it
- ResNet101 finetuned till block 4

Chosen by the chromosome [0, 1, 1, 1, 1, 0, 0, 1, 0, 0, 0, 0], our winner in the final population.

5.1.5 GA Results for ResNet Inception

The following graph shows the results obtained for each generation from the population (i.e., the maximum and mean accuracy):

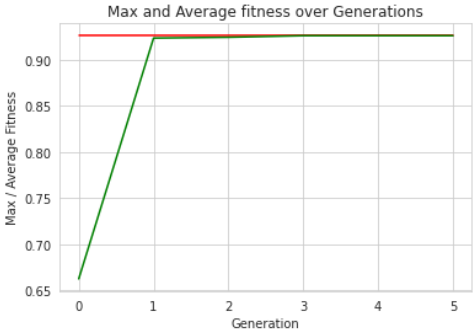


Figure 37: Maximum and average accuracy in the population for each generation.

The maximum accuracy value for Inception was obtained in Section 4.4.5 fine tuning 3 blocks and using exponential decay for the learning rate, reaching an accuracy score of 0.5803. Also here we exceed the value reached by the single model obtaining a much higher value, reaching 0.9264, satisfactory value even if lower than the one obtained with the ensemble of VGGs. This was obtained from the genetic algorithm by combining the following models together:

- Inception with 1 block finetuned
- Inception with 3 blocks finetuned

Chosen by the chromosome [0, 1, 0, 1, 0], our winner in the final population.

6 — Conclusion

The project reports a series of possible solutions for the classification of tumors from a given dataste. The solutions are therefore based on the use of multiple CNN networks, considering the current state of the art and the limitations of Colab.

FRASI SULLA SCRATCH

Regarding the pre-trained models we were able to obtain more satisfactory results than scratch and the best results were obtained thanks to VGG16, which with its simplicity manages better to classify than the larger networks that are trained with greater accuracy on *imagenet*, which has a dataset very different from ours and perhaps this is the reason why we obtained slightly worse results for ResNets and Inception. Much more satisfactory results have been obtained thanks to the ensemble, reaching thanks to the aggregation of VGG16s models about 96% of accuracy, the highest value reached in all our tests, bringing the model obtained to be our final model for the identification of the painters of the paintings. One thing to keep in mind talking about the ensemble is the fact that the genetic algorithm finds the conclusion immediately or almost immediately, this thing is related to a lucky starting point, but also, above all, to the very small number of genes that are part of the different chromosomes. As for the state-of-art, we said in the introduction that the highest value obtained on the test accuracy was found by Nitin Viswanathan, and it was about 90%, just few points below us. This makes our final result higher than the state-of-the-art.